PROGRAMMER'S-
GUIDE

GFA-BASIC
PC

MS-DOS 80×86

The publisher has made every effort to publish the complete and accurate information. GFA Systemtechnik asumes no responsibility that the described procedures, programs etc. are functional and free from third party rights.

Program authors: Frank Ostrowski
Roland Schütz
Dirk van Assche
Michael Wiegand

Handbook authors: Dr. Wolfgang Buscher
Frank Ostrowski

Translation: Donald P. Maple

1. Edition, December 1990

Table of Contents

1. Some programming suggestions .... 1 - 1

1.1 Loops .... 1 - 2
1.1.1 FOR...NEXT loop .... 1 - 2
1.1.2 WHILE...WEND loop .... 1 - 6
1.1.3 REPEAT...UNTIL loop .... 1 - 7
1.1.4 DO...LOOP loop .... 1 - 8

1.2 Conditional statements .... 1 - 10
1.2.1 IF...ENDIF conditional statement .... 1 - 10
1.2.2 SELECT (SWITCH)...CASE conditional statement . 1 - 14

1.3 Writing subroutines .... 1 - 19
1.3.1 Call by Value- und Call by Reference variablen .... 1 - 22
1.3.2 Programming hints .... 1 - 25
1.3.3 Single line functions .... 1 - 33

2. Special commands and functions .... 2 - 1

2.1 The variables .... 2 - 1
2.1.1 Double .... 2 - 1
2.1.2 Long .... 2 - 2
2.1.3 Word .... 2 - 2
2.1.4 Byte .... 2 - 2
2.1.5 Boolean .... 2 - 3
2.1.6 String .... 2 - 3

2.2 Operators .... 2 - 5
2.2.1 Floating point operatos .... 2 - 5
2.2.2 Integer operators .... 2 - 8
2.2.3 Logical operators .... 2 - 8
2.2.4 Bitwise operators .... 2 - 10
2.2.5 Combinding strings .... 2 - 14

2.2.6 Relational operators .... 2 - 14
2.2.7 Operator hierarchy .... 2 - 17

2.3 Type conversion .... 2 - 18
2.3.1 Conversion of numeric into string expressions .... 2 - 18
2.3.2 Conversion of strings into numerical expressions .... 2 - 23

2.4 Using strings .... 2 - 26
2.4.1 The String operators .... 2 - 28
2.4.2 The Manipulation functions .... 2 - 29
2.4.3 The Evaluation functions .... 2 - 33

2.5 The MAT commands .... 2 - 35
2.5.1 The system command MAT BASE .... 2 - 35
2.5.2 The generating commands .... 2 - 39
2.5.3 The read and writing commands .... 2 - 45
2.5.4 The copy and exchange commands .... 2 - 48
2.5.5 The operating commands .... 2 - 57

2.6 Graphics .... 2 - 103
2.6.1 Text graphic cards .... 2 - 103
2.6.1.1 The monochrome (IBM) video card (MDA) .... 2 - 104
2.6.1.2 The monochrome (Hercules) graphic card (HGA) . 2 - 104
2.6.1.3 The (IBM) colour graphic card (CGA) .... 2 - 104
2.6.2 The commands and functions for character graphics .... 2 - 106
2.6.3 The Pixel graphic card .... 2 - 113
2.6.3.1 The EGA card .... 2 - 114
2.6.3.2 The VGA card .... 2 - 114
2.6.4 The commands and functions for pixel graphics .... 2 - 115

3.	**The commands and functions for creation of a (pixel-) graphic interface**	**3 - 1**
3.1	**Menu bars**	**3 - 1**
3.2	**Alert boxes and Pop-up menus**	**3 - 16**
3.2.1	ALERT	3 - 16
3.2.2	POPUP	3 - 21
3.3	**Windows**	**3 - 27**
3.3.1	Redraw events	3 - 56
4.	**System routines**	**4 - 1**
4.1	**Keyboard**	**4 - 2**
4.2	**Serial port**	**4 - 5**
4.3	**Printer (parallel) port**	**4 - 10**
4.4	**Joystick**	**4 - 13**
4.5	**Read disk sectors**	**4 - 15**
	DOS Interrups	**4 - 17**
	Index	

1. Some programming suggestions

Structured programming is a functional term implying clearly arranged program code and program flow based on certain structures. Among these structure we include:

1. **Loops**
2. **Conditional statements**
3. **Subroutines**

The correct usage of these structures enhances the clarity of a program and is an important requirement for program optimisation in regards to code length and/or speed of execution.

1.1 Loops

The loops are used whenever a program segment is executed repeatedly. Formally, a loop comprises the loop statement and the loop contents. The loop contents is the program segment which is to be executed repeatedly. The loop statement is composed of the loop header and the loop footer. Various loops implemented in GFA-BASIC are differentiated by distinct structures in the loop header and loop footer.

1.1.1 FOR...NEXT loop

The best known loop is the **FOR...NEXT** loop. It's a counting loop, whose header contains the condition for loop start and loop end while the footer in/de-crements a variable.

The formal definition of the **FOR...NEXT** loop is as follows:

```
FOR variable=loopstart TO loopend [STEP interval]
  program segment
NEXT variable
```

The variable can be any numeric variable. However, for maximum speed it's advisable to use integer variables whenever possible. **FOR...NEXT** loops are used mainly for manipulation of arrays and for graphic output.

Example:

You wish to initialise an array with particular values. The loop to do this could look as follows:

```
DIM a(10)
DATA 1.2,3,4.2,5,6.4,2,7.8,9.1,1,3,6.4
FOR i%=0 TO 10
  READ a(i%)
NEXT i%
```

In this example the loop counter i% in loop footer is incremented by 1 until the first time it becomes greater than the loop criterion (i% = 10). When the loop ends the loop counter is 1 higher than the given loop criterion.

By using **STEP**, the amount by which the loop counter is in/de-cremented can be changed. If **STEP** is followed by a negative value the loop counter is decremented, while a positive value will increment it.

```
FOR i%=100 TO 0 STEP -2
  program segment
NEXT i%
```

causes the loop counter i% to be decremented by 2, during each loop pass,

```
FOR i%=0 TO 100 STEP 2
  program segment
NEXT i%
```

causes the loop counter i% to be incremented by 2, during each loop pass.

By using **STEP** it's also possible to specify fractional values:

```
FOR i=0 TO 1 STEP 0.1
  program segment
NEXT i
```

In this case the loop counter can no longer be an integer variable. However, for achieving maximum speed you should always attempt to write **FOR...NEXT** loops using integer variables. The loop in the above example executes eleven times. If only the number of loop runs is important, the loop can be modified as follows::

```
FOR i%=0 TO 10
  program segment
NEXT i%
```

But if the program segment uses the loop counter (for example, to scale a graph) the two loops are no longer equivalent. However, they can be made to act the same if the difference in the loop counter is accounted for during the scaling. So, for example, the loops

```
FOR i=0 TO 1 STEP 0.1
  MUL a,i
NEXT i
```

and

```
FOR i%=0 TO 10
  MUL a,i%/10
NEXT i%
```

are completely identical. But, the second loop is faster by about a factor of 10.

Formally, the loops can be separated into entry-tested and exit-tested. As a rule of thumb, exit-tested loops execute at least once, while the entry-tested loops may, under certain conditions, not execute at all. Whether a loop is entry-tested or exit-tested depends on whether the loop condition is tested in the loop header or in the loop footer. If the test occurs in the header the loop is entry-tested, otherwise it's exit-tested.

The **FOR...NEXT** loop is entry-tested, since the test whether the loop start is less (in case of positive values) or greater (in case of negative values) than the loop end, occurs in the loop header.

If a loop should end before the loop criterion is satisfied, an **EXIT IF** condition should be incorporated within the body of the loop, i.e. the program segment. In general, a loop may contain any number of **EXIT IF** conditions.

Example:

You wish to write a program to test that no element in an array is equal to zero. To do this you can use the following loop:

```
DIM a(100)
FOR i%=0 TO 100                    //      initialise array
  a(i%)=3-RAND(10)
NEXT i%
FOR i%=0 TO 100
  EXIT IF a(i%)=0
NEXT i%
IF i%=101
  PRINT "no element is equal to 0"
ELSE
  PRINT "a(";i%;")=0"
ENDIF
```

The **EXIT IF** command is a way to exit from a loop in a controlled fashion. Other BASIC dialects would use:

```
FOR i%=0 TO 100
  IF a(i%)=0 THEN EXIT FOR
NEXT i%
```

You can use this format in GFA-BASIC as well. The interpreter will automatically convert it to

```
FOR i%=0 TO 100
  EXIT IF a(i%)=0        // FOR
NEXT i%
```

The **FOR...NEXT** loops can be nested within each other at will:

```
DIM a(10,10,10)
FOR i%=0 TO 10
  FOR j%=0 TO 10
    FOR k%=0 TO 10
      a(i%,j%,k%)=RAND(100)
    NEXT k%
  NEXT j%
NEXT i%
```

However, attention should be paid to match up all **FOR** and **NEXT** statements. This means that the last **FOR** should use the same counter as the first **NEXT**, and that the first **FOR** should use the same counter as the last **NEXT**.

The loops are mostly used in situations where the simple incrementing or decrementing of a variable is not enough. The following loops are available in GFA-BASIC: **WHILE...WEND**, **REPEAT...UNTIL** and the **DO...LOOP** group of loops.

1.1.2 WHILE...WEND loop

The **WHILE...WEND** is an entry-tested loop with the following formal definition:

```
WHILE condition true
  program segment
WEND
```

It runs for as long as the condition after the **WHILE** in the loop header is logically true.

Example:

```
WHILE MOUSEK AND 1     // draws at the current mouse
//                        position
  PLOT MOUSEX,MOUSEY   // for as long as the left
WEND                   // mouse button is held down

a$=""
WHILE a$<>"Hello GFA"  // the loop runs until
  INPUT "Enter a greeting";a$//      Hello GFA is
  //                                 entered.
WEND
```

1.1.3 REPEAT...UNTIL loop

The **REPEAT...UNTIL** is an exit-tested loop with the following formal definition:

```
REPEAT
  program segment
UNTIL condition true
```

It runs for as long as the condition after the **UNTIL** in the loop footer is logically true.

Example:

```
REPEAT                          //      the loop runs
  INPUT "Enter a greeting";a$//         until Hello
UNTIL a$="Hello GFA"            //      GFA is entered.
```

The **WHILE...WEND** and **REPEAT...UNTIL** loops test logically complementary conditions. This means that each **WHILE...WEND** loop can be converted to a **REPEAT...UNTIL** loop, and so logically negate the **WHILE** condition.

Example:

```
WHILE NOT EOF
  INPUT #1,a$
WEND
```

is equivalent to

```
IF LOF(#1)
  REPEAT
    INPUT #1,a$
  UNTIL EOF
ENDIF
```

The **IF** statement before the **REPEAT** is necessary here, since the **EOF** can be reached after the very first **INPUT**. This would cause **INPUT #1** to report an error.

The most elegant solution would be to use the **DO...LOOP**:

```
DO UNTIL EOF
  Input #1,a$
LOOP
```

More about it later.

In general, you should use the loop which doesn't contain a logical negation in its criterion, since this is not only clearer but also faster.

As in all other loops, you can exit from both **WHILE...WEND** and **REPEAT...UNTIL** in a controlled fashion by using the **EXIT IF** condition.

1.1.4 DO...LOOP loop

The **DO...LOOP** in GFA-BASIC has undergone a number of changes to improve the program structure. The basic **DO...LOOP** is an endless loop which can be terminated in a controlled fashion only by using **EXIT IF**.

Example:

```
DO
  MOUSE mx%,my%,mk%
  IF mk%
    PLOT mx%,my%
  ENDIF
LOOP
```

This endless loop draws points at the current mouse position when the mouse button is pressed.

In GFA-BASIC it's possible to combine the **DO...LOOP** loop with the test structures of **WHILE...WEND** and **REPEAT...UNTIL**. The following forms of the loop are used for this purpose:

```
DO ... LOOP
DO ... LOOP UNTIL
DO ... LOOP WHILE
DO ... WEND
DO ... UNTIL

WHILE ... LOOP          DO WHILE ... LOOP
WHILE ... LOOP UNTIL    DO WHILE ... LOOP UNTIL
WHILE ... LOOP WHILE    DO WHILE ... LOOP WHILE
WHILE ... WEND          DO WHILE ... WEND
WHILE ... UNTIL         DO WHILE ... UNTIL

REPEAT ... LOOP         DO UNTIL ... LOOP
REPEAT ... LOOP UNTIL   DO UNTIL ... LOOP UNTIL
REPEAT ... LOOP WHILE   DO UNTIL ... LOOP WHILE
REPEAT ... WEND         DO UNTIL ... WEND
REPEAT ... UNTIL        DO UNTIL ... UNTIL
```

1.2 Conditional statements

During the execution of a program there are many places where the status of a variable or an expression determines the rest of the run. In such a case the so-called conditional statements are used. They cause the program to execute in a certain way depending on the condition of a variable or an expression.

The following conditional statements are available in GFA-BASIC:

IF...ENDIF

and

SELECT...CASE

1.2.1 IF...ENDIF conditional statement

The **IF...ENDIF** conditional statement has the following formal definition:

start of conditional statement

IF condition

Program segment, which is executed if the condition is fulfilled (is logically 'true'). This is followed by the branch to the end of the conditional statement.

ELSE IF condition

Program segment, which is executed if the condition is fulfilled (is logically 'true'). This is followed by the branch to the end of the conditional statement.

ELSE

Program segment, which is executed if none of the previous conditions were fulfilled. This is followed by a branch to the end of the conditional statement.

ENDIF

end of conditional statement

This conditional statement must always have the **IF** and **ENDIF** components. The **ELSE IF** and **ELSE** are optional, i.e. they are used only when there is a variety of conditions. Each **IF...ENDIF** conditional statement can contain only one **IF**, one **ELSE** and one **ENDIF**, however, any number of **ELSE IF** branches are allowed. The **IF...ENDIF** conditional statement is used generally much more than the **SELECT...CASE** described below, since the conditions after **IF** or **ELSE IF** can be completely independent of each other.

How to use the IF...ENDIF conditional statements:

Example IF...ENDIF:

You wish to write a program in which the value of the variable b depends on variable a. The condition is that the sign of b must be changed if a is negative. You can use the following form of the **IF...ENDIF** statement:

```
IF a < 0
  MUL b,-1
ENDIF
```

Example IF...ELSE IF...ENDIF:

You wish to write a program in which the value of the variable b depends on variable a. The condition is that the sign of b must be changed if a is greater than 100 or less than 10. You can use the following form of the **IF...ENDIF** statement:

```
IF (a>100) OR (a<10)
  MUL b,-1
ENDIF
```

You can also use the following:

```
IF a>100
  MUL b,-1
ELSE IF a<10
  MUL b,-1
ENDIF
```

In the above case the logical comparison in the first **IF** is split up by using an additional **ELSE IF**. In most cases this results in faster execution.

Example IF...ELSE IF...ELSE...ENDIF:

You wish to write a program in which the value of the variable b depends on variables a or c. The condition is that the sign of b must be changed if a is greater than 100 or c is less than 10. Otherwise b should be positive. You can use the following form of the **IF...ENDIF** statement:

```
IF a>100
  MUL b,-1
ELSE IF c<10
  MUL b,-1
ELSE
  b=ABS(b)
ENDIF
```

The **IF...ENDIF** conditional statements can be nested within each other at will.

Example IF...IF...ENDIF...ENDIF

You wish to write a program in which the value of the variable b depends on variables a and c. The condition is that the sign of b must be changed if a is greater than 100 or c is less than 10. Otherwise b should be positive. You can use the following form of the **IF...ENDIF** statement:

```
IF (a>100) AND (c<10)
  MUL b,-1
ELSE
  b=ABS(b)
ENDIF
```

The logical comparison in the **IF condition** can be split up by nesting two **IF** statements:

```
IF a>100
  IF c<10
    MUL b,-1
  ELSE
    b=ABS(b)
  ENDIF
ELSE
  b=ABS(b)
ENDIF
```

The following example contains a very complex conditional branch. You wish to write a program in which various subroutines are invoked depending on the mouse position and on whether the left mouse button is pressed. The first prerequisite for a branch to a subroutine is that the left mouse button is pressed. If it is, a branch to subroutine 1 should be taken when the mouse pointer is within the screen coordinates 0,0,100,100. A branch to subroutine 2 should be taken when the mouse pointer is within

the screen coordinates 101,0,200,100. The subroutines 3 and 4 should be run when the mouse pointer is within the screen coordinates 0,101,100,200 and 101,101,200,200 respectively. The conditional block to do this may look as follows:

```
IF MOUSEK AND 1                  //      left mouse button pressed
  IF MOUSEX <=100                //      subroutine 1 or 3
    IF MOUSEY<=100               //      subroutine 1
      @1
    ELSE
      IF MOUSEY <=200            //      subroutine 3
        @3
      ENDIF
    ENDIF
  ELSE
    IF MOUSEX <=200              //      subroutine 2 or 4
      IF MOUSEY<=100             //      subroutine 2
        @2
      ELSE
        IF MOUSEY <=200          //      subroutine 4
          @4
        ENDIF
      ENDIF
    ENDIF
  ENDIF
ENDIF
```

1.2.2 SELECT(SWITCH)...CASE conditional statement

The second conditional statement, **SELECT...CASE**, is used only when special conditions must be tested for. In such a case it offers considerable speed gains over the **IF...ENDIF** statement. The special conditions are limited to the following:

a) the **CASE** branch in the conditional statement is always related to the value of a variable,

b) this variable is an integer variable or a string variable - up to first four characters of which are significant - and

c) the **CASE** branch can't contain any direct logical comparisons.

The **SELECT...CASE** conditional statement has the following formal definition:

start of conditional statement

SELECT integer variable or string

CASE condition

program segment, which is executed if the integer or string variable specified after **SELECT** fulfils the condition after **CASE**. If the end of the program segment contains no **CONT** a branch is taken to the end of the conditional statement. If the program segment is followed by a **CONT**, the next **CASE** or **DEFAULT** is executed.

DEFAULT

program segment, which is executed if the integer or string variable specified after **SELECT** does not fulfil any of the conditions after **CASE** or if the last **CASE** before **DEFAULT** was processed and its program segment ends with a **CONT**. This is followed by a branch to the end of the conditional statement.

ENDSELECT

end of conditional statement

This conditional statement must always have the **SELECT** component, at least one **CASE** and one **ENDSELECT**. Further **CASE**, as well as **DEFAULT** components, are optional, i.e. they are used only when there is a variety of conditions. Each **SELECT...CASE** conditional statement can contain only one **SELECT**, one **DEFAULT** and one **ENDSELECT**, however, any number of **CASE** components is allowed. Several conditions can

follow **CASE** but they must be separated by commas. By using **TO**, it is also possible to declare a range of values for **CASE**.

Instead of **SELECT**, **SWITCH** can be used also. If it is, the conditional statement must be terminated with an **ENDSWITCH**.

How to use SELECT...CASE conditional statements:

Example SELECT...CASE

You wish to write a program in which the integer variable a% can assume various values. When a%=10 the subroutine 1 should be executed and when a%=20 the subroutine 2 should be invoked. The conditional statement to do this could then look as follows:

```
SELECT a%
CASE 10
  @1
CASE 20
  @2
ENDSELECT
```

Example SELECT...CASE...CONT

You wish to write a program in which the integer variable a% can assume various values. If a% is between 0 and 10 the subroutine 1 should be invoked and if a% is between 5 and 15 subroutine 2 should be executed. For values 6, 7, 8, 9 and 10 both subroutine 1 as well as subroutine 2 should run. The conditional statement to do this could then look as follows:

```
SELECT a%
CASE 0,1,2,3,4,5,6,7,8,9,10
  @1
  CONT
CASE 5,6,7,8,9,10,11,12,13,14,15
```

```
@2
ENDSELECT
```

In this case, however, it's much more elegant to specify a range after **CASE** by using **TO**:

```
SELECT a%
CASE 0 TO 10
  @1
  CONT
CASE 5 TO 12
  @2
ENDSELECT
```

Both range and individual values can be combined:

```
SELECT a%
CASE 0,1,2,3 TO 5,6 TO 10       //    integer values from 0 to 10
CASE TO 10, 23,30               //    integer values until 10 as well
//                                    as the values 23 and 30
CASE 10,11,30 TO                //    the values 10 and 11 as well as
                                //    integer values from 30
```

The **SELECT...CASE** conditional statements can be nested within each other to any level.

SELECT...CASE and **IF...ENDIF** conditional statements can be mixed together and nested to any level.

Example IF...SELECT...CASE...ENDIF

For an example of how to combine the **IF...ENDIF** and **SELECT...CASE** statements the above program to monitor the mouse position and the pressing of the left mouse button will be used again.

The first prerequisite for a branch to a subroutine was the pressing of the left mouse button. If the button is pressed, a branch to subroutine 1 should be taken when the mouse pointer is within the screen coordinates 0,0,100,100.

A branch to subroutine 2 should be taken when the mouse pointer is within the screen coordinates 101,0,200,100.

The subroutines 3 and 4 should be run when the mouse pointer is within the screen coordinates 0,101,100,200 and 101,101,200,200 respectively.

The conditional block to do this may look as follows:

Example:

```
IF MOUSEK AND 1          // left mouse button pressed
  SELECT MOUSEX
  CASE TO 100            // subroutine 1 or 3
    SELECT MOUSEY
    CASE TO 100          // subroutine 1
      @1
    CASE 101 TO 200      // subroutine 3
      @3
    ENDSELECT
  CASE 101 TO 200        // subroutine 2 or 4
    SELECT MOUSEY
    CASE TO 100          // subroutine 2
      @2
    CASE 101 TO 200      // subroutine 4
      @4
  ENDSELECT
ENDIF
```

It's easy to see that the program has become substantially shorter and easier to read. Furthermore, the program now runs about five times as fast.

Whenever possible, **SELECT...CASE** should always be used instead of **IF...ENDIF**. The compiler will optimise it by producing shorter code and it will run faster.

1.3 Writing subroutines

Writing subroutines is one of the most important component of structured programming. The subroutines are used whenever a program contains interdependent parts, which are called by several program segments and/or by one program segment several times. One of the simplest, but very frequently used subroutines, is the waiting loop. It's used whenever the user is informed about something by using an event (screen output, for example). A waiting loop can, for example, look like this:

Example 1:

```
FOR i%=1 to 5000
NEXT i%
```

Example 2

```
REPEAT
UNTIL MOUSEK
```

Example 3

```
DO
  a%=ASC(INKEY$)
UNTIL a%=27
```

The waiting loop in **Example 1** consists of an "empty" loop executed 5000 times. This is one of the oldest waiting loops and is today used only very rarely. This is mainly because the waiting time is not constant and depends on the processor (8086, 80286, 8036) and its speed.

The waiting loop in **Example 2** runs until the user clicks the mouse button. Obviously, this solution is only applicable when the user has a mouse.

The waiting loop in **Example 3** runs until the user types in the 'Esc' key. Depending on which ASCII code is tested for in the **UNTIL** condition, this loop can be used in a general way and is fairly standard.

If you use several waiting loops in your program it makes sense to define a subroutine that contains this waiting loop. It could be done as follows:

Example:

```
PROCEDURE wait
  LOCAL a%
  DO
    a%=ASC(INKEY$)
  LOOP UNTIL a%=27
RETURN
```

A **PROCEDURE** is used here to define the subroutine. The main program should then - instead of the waiting loop - only use the procedure called 'wait', '@wait' or '**GOSUB** wait'.

GFA-BASIC uses two types of subroutines:

Procedures:	**PROCEDURE**	and
Functions:	**FUNCTION**	

The most important difference between the two types is that the functions always return a value while the procedures never return a value.

This difference is easy to see in the formal definition of the two types:

```
PROCEDURE[(parameter list)]
program segment
RETURN .
```

and

```
FUNCTION[(parameter list)]
program segment
  RETURN value
ENDFUNC
```

The basic definition of a function contains therefore at least one **RETURN** value. It is also possible for a **FUNCTION** to contain several **RETURN** value lines. This is mostly the case when the **FUNCTION** contains a conditional block:

Example:

```
FUNCTION evaluate$(a%)
  SELECT a%
  CASE TO 0
    RETURN "less than or equal to 0"
  CASE 1 TO 10
    RETURN "between 1 and 10"
  CASE 11 TO 50
    RETURN "between 11 and 50"
  DEFAULT
    RETURN "greater than 50"
  ENDSELECT
ENDFUNC
```

When writing a program you should always be careful to use the correct subroutine type for each problem. In principle, you can convert each **PROCEDURE** into a **FUNCTION** (and vice versa) but this may not always make sense.

As a rule of thumb:

The **FUNCTION**s are normally used in all types of calculations, while the **PROCEDURE**s are used for controlling the program flow.

To facilitate their optimum usage, the subroutines in GFA-BASIC can be passed variables and/or arrays. These are specified in the subroutine parameter list. Furthermore, local variables can be defined within the subroutines.

```
@example1(3,5,a%,b(),a$())
PROCEDURE example1(v1,v2,w%, VAR f1(),f$())
  LOCAL i%,j%
  program segment
RETURN
```

or

```
@example2(3,5,a%,b(),a$())
FUNCTION example2(v1,v2,w%, VAR f1(),f$())
  LOCAL i%,j%
  program segment
  RETURN value
ENDFUNC
```

1.3.1 Call-by-value and Call-by-reference variables

There are two categories of variables which can be passed to a subroutine:

1. variables, whose value is passed and
2. variables, which are themselves passed.

The first category is also known as **'Call by value'** and the second as **'Call by reference' variables**.

The most important difference between the two can be seen in an example (from Engels, G.P. & Görgens, M.C., GFA-BASIC Version 3.0; GFA Systemtechnik, 1988):

Example:

```
b$="ABS123ABS"
PRINT "Original string:"'b$
PRINT
PRINT "Call by value:"
@call_by_value(b$)
PRINT "The calling program receives"'b$'"back"
```

```
PRINT
PRINT "Call by reference:"
@call_by_reference(b$)
PRINT "The calling program receives"'b$'"back"

PROCEDURE call_by_value(a$)
  LOCAL i&,code|
  FOR i&=1 TO LEN(a$)
    code|=ASC(MID$(a$,i&,1))
    IF code|<65 OR code|>90
      MID$(a$,i&)=" "
    ENDIF
  NEXT i&
  PRINT "Received:"'a$
RETURN

PROCEDURE call_by_reference(VAR a$)
  LOCAL i&,code|
  FOR i&=1 TO LEN(a$)
    code|=ASC(MID$(a$,i&,1))
    IF code|<65 OR code|>90
      MID$(a$,i&)=" "
    ENDIF
  NEXT i&
  PRINT "Received:"'a$
RETURN
```

This program produces the following output:

```
Original string: ABC123ABC
Call by value:
Returns: ABC   ABC

Call by reference
Returns: ABC   ABC
```

The main program passes the same string ABC123ABS to both call_by_value and call_by_reference subroutines. Both subroutines search this string for characters whose ASCII code is less than 65 or greater than 90, i.e. for characters which are not upper case. If the subroutine finds such a character it replaces it with a space. Both subroutines will therefore replace the 123 in the original string ABC123ABC with spaces. Both subroutines will also produce the same message:

```
Returns: ABC   ABC
```

The string returned to the main program by the subroutines is also interesting. The call_by_value subroutine returns ABC123ABC while the call_by_reference subroutine returns ABC   ABC.

In case of 'Call by value' category the variable itself is not modified, while 'Call by reference' category does indeed modify it. In other words, 'Call by value' variables are treated by the subroutine as local variables while 'Call by reference' variables are treated by the subroutine as global variables.

The 'Call by reference' variables need the prefix **VAR** in the parameter list. Should both 'Call by value' and 'Call by reference' variables be passed to a subroutine, the parameter list must first name the 'Call by value' variables and then - prefixed by **VAR** and separated by commas - the 'Call by reference' variables. The arrays can only be passed as 'Call by reference' variables!

Another feature of subroutines is that they can use local variables. GFA-BASIC takes them literally and recognizes them only within the subroutine in which there were declared by using **LOCAL**. If a subroutine calls another subroutine, the local variables in the calling program are not known to the called program. They must be passed to the called subroutine as 'Call by value' or as 'Call by reference' variables.

1.3.2 Programming hints

You should familiarize yourself with various variable types. In general, a program can contain six different variable categories:

1. global variables
2. 'Call by value' variables
3. 'Call by reference' variables
4. local Variables
5. variables, which belong to a **TYPE** definition
6. variables, defined during debugging of a program which are deleted afterwards.

Out of all these variables only the **TYPE** defined variables stand out (because of the point). The labels for all other variable categories are freely selectable. To make programs more readable for future changes and/or corrections it makes sense to clearly label the variables in all other categories.

For **debug variables** (category 6) we recommend the doubling of the variable names:

```
aa%, ii%,kk$...
```

For **local variables** (category 4) we use an underdash at the end of the variable name:

```
a_%, i_%, k_$...
```

The **global variables** are not particularly marked:

```
a%, i%, k$...
```

The **'Call by value' variables** are marked by appending _v, while 'Call by reference' variables are marked by appending _r:

```
a_v%, i_v%, k_v$...        'Call by value' variables
a_r%, i_r%, k_r$...        'Call by reference' variables.
```

One of the most useful applications of subroutines are the 'word lists'. Word lists contain the complete text for interaction between the program and the user. If all those word lists are assembled together, they're very easy to change, expand, translate etc., without having to first search for them throughout the program. As an example of how to construct such subroutines here's a listing which contains various strings for the '**ALERT**' messages:

Example:

```
PROCEDURE alert_list(m_v% VAR r_r%)
  LOCAL f_!,i_%,j_%,a_$,b_$,e_$,r_$
  f_!=TRUE
  e_$="Enter "
  r_$="CANCEL"
  SELECT m_v%
  CASE 1
    i_%=1
    a_$=e_$+"not found"
    j_%=1
    b_$=r_$
  CASE 2
    i_%=2
    a_$=e_$+"is multiply defined"
    j_%=2
    b_$="CONTINUE|"+r_$
  CASE 3
    i_%=1
    a_$=e_$+"is protected"
    j_%=1
    b_$=r_$
```

```
CASE 4
  i_%=3
  a_$=e_$+"is not|complete"
  j_%=1
  b_$="CONTINUE|"+r_$
CASE 5
  i_%=3
  a_$=e_$+"delete"
  j_%=2
  b_$="YES|NO"
DEFAULT
  f_!=FALSE          // for unforeseen cases
  r_r%=-1
ENDSELECT
IF f_!
  ALERT i_%,a_$,j_%,b_$,r_r%
ENDIF
RETURN
```

The calling program need only pass to the subroutine the number of the selected message in m_v% and it will receive back the chosen message in r_r%.

In a similar fashion, you can also create subroutines which contain the word lists for **INPUT**, **PRINT** and **TEXT** output.

Another possible application for a subroutine may be the analysis of numeric arrays:

```
PROCEDURE chk_dim(VAR array_r(),z_r%,s_r%,dim_r%)
  LOCAL v_%
  //
  //  depending on OPTION BASE 0 or 1 */
  //
  ERASE check_option_base&()
  DIM check_option_base&(1)
  v_%={{*check_option_base&()}}-1 //     v_%=1 for OPTION BASE 0;
```

```
  //                                      otherwise 0
  ERASE check_option_base&()
  //
  dim_r%=INT{*array_r()+4}        //      returns the dimension
  //                                      of the array
  SELECT dim_r%                   //      only one and two dimensional
  //                                      arrays are examined
  //
  CASE 1                          //      one dimensional array
    //
    z_r%={{*array_r()}}-v_%       //      number of rows in array
    s_r%=1                        //      number of columns in array
    //
  CASE 2                          //      two dimensional array
    //
    z_r%={{*array_r()}+4}-v_%     //      number of rows in array
    s_r%={{*array_r()}}-v_%       //      number of columns in array
    //
  DEFAULT                         //      a message in case of
  //                                      an array which is not
  //                                      one or two dimensional
  ENDSELECT
RETURN
```

This program is useful for **BMOVE** which moves complete blocks of values from one array to another.

The consistent use of subroutines and adherence to variable classification enables you to program routines which are designed as modules and can be incorporated in many different programs. The module Statistic can serve as an example of this as it combines several smaller subroutines to calculate various statistical values:

calculation of the arithmetic mean

```
//
FUNCTION arimean(n_v%,VAR vector_r())
  LOCAL i_%,xq_
  IF n_v%<1
    RETURN code
  ENDIF
  //   Code is a global variable, which must be assigned in the calling
  //   program. As a rule, code is set to a fixed value, for example
  //   1.7777777E-13, and then tested if it corresponds to the value
  //   returned from FUNCTION. If it does, an error message is reported.
  //
  FOR i_%=1 TO n_v%
    ADD xq_,vector_r(i_%)
  NEXT i_%
  RETURN xq_/n_v%
ENDFUNC
//
```

calculation of the geometric mean

```
//
FUNCTION geomean(n_v%,VAR vector_r())
  LOCAL no_!,i_%,xq_
  xq_=1
  //
  FOR i_%=1 TO n_v%
    IF vector_r(i_%)=>0
      MUL xq_,vector_r(i_%)
    ELSE
      no_!=TRUE     //          a value < 0 was found
    ENDIF
    EXIT IF no_!    //          geometric mean for values =>0 only
  NEXT i_%
  //
  IF NOT no_!
    RETURN xq_^(1/n_v%)
```

```
  ENDIF
  RETURN code       //          for description of code see arimean
ENDFUNC             //          FUNCTION
//
```

calculation of the harmonic mean

```
//
FUNCTION harmean(n_v%,VAR vector_r())
  LOCAL no_!,i_%,xq_
  //
  FOR i_%=1 TO n_v%
    IF vector_r(i_%)
      ADD xq_,1/vector_r(i_%)
    ELSE
      no_!=TRUE     //          a value <> 0 was found
    ENDIF
    EXIT IF no_!    //          harmonic mean for values <> 0 only
  NEXT i_%
  //
  IF xq_>0
    IF NOT no_!
      RETURN n_v%/xq_
    ENDIF
  ENDIF
  RETURN code       //          for description of code see arimean
ENDFUNC             //          FUNCTION
//
```

calculation of the empirical median

```
//
FUNCTION median(n_v%,VAR vector_r())
  LOCAL xq_
  DIM h_(n_v%)
  BMOVE V:vector_r(1),V:h(0),SHL(n_v%,3)
  SSORT h_(),n_v%
  INSERT h_(0)=0
  IF ODD(n_v%)
    xq_=h_(SUCC(n_v%)/2)
  ELSE
```

```
    xq_=(h_(n_v%/2)+h_(SUCC(n_v%/2)))/2
  ENDIF
  ERASE h_()
  RETURN xq_
ENDFUNC
//
calculation of the empirical variation
//
FUNCTION variation(n_v%,VAR vector_r())
  LOCAL i_%,xq_,xv_
  //
  IF n_v%<2
    RETRUN code      //           for description of code see arimean
  ENDIF              //           FUNCTION
  //
  xq_=@arimean(n_v%,vector_r())
  //
  FOR i_%=1 TO n_v%
    ADD xv_,(vector_r(i_%)-xq_)^2
  NEXT i_%
  //
  RETURN xv_/PRED(n_v%)
ENDFUNC
//
calculation of the empirical standard deviation
//
FUNCTION std(n_v%,VAR vector_r())
  LOCAL xv_
  //
  xv_=@variation(n_v%,vector_r())
  IF xv_<>code
    RETURN SQR(xv_)
  ENDIF
  RETURN code        //           for description of code see arimean
ENDFUNC              //           FUNCTION
//
```

```
calculation of the empirical standard error
//
FUNCTION ste(n_v%,VAR vector_r())
  LOCAL xv_
  //
  xv_=@std(n_v%,vector_r()
  IF xv_<>code
    RETURN xv_/SQR(n_v%)
  ENDIF
  RETURN code        //          for description of code see arimean
ENDFUNC              //          FUNCTION
//
calculation of the empirical median deviation
//
FUNCTION mediandev(n_v%,VAR vector_r())
  DIM hh(n&)
  LOCAL median,medev
  median=@median(n&,daten())
  FOR iii%=1 TO n&
    hh(iii%)=ABS(daten(iii%)-median)
  NEXT iii%
  medev=@median(n&,hh())
  ERASE hh()
  RETURN medev
ENDFUNC
```

1.3.3 Single line functions

A special form of subroutines are single line functions. They are mainly created to define mathematical functions. As an example here are a few hyperbolean functions and their reverse area functions.

Sinus hyperbolicus:

```
DEFFN sinh(x_r)=(EXP(x_r)-EXP(-x_r))/2
```

Cosinus hyperbolicus:

```
DEFFN cosh(x_r)=(EXP(x_r)+EXP(-x_r))/2
```

Tangens hyperbolicus:

```
DEFFN tanh(x_r)=(1-EXP(-2*x_r))/(1+EXP(-2*x_r))
```

Cotangens hyperbolicus:

```
DEFFN coth(x_r)=(1+EXP(-2*x_r))/(1-EXP(-2*x_r))
```

Area Sinus hyperbolicus:

```
DEFFN arsinh(x_r)=LOG(x_r+SQR(x_r^2+1))
```

Area Cosinus hyperbolicus:

```
DEFFN arcosh(x_r)=LOG(x_r+SQR(x_r^2-1))
```

Area Tangens hyperbolicus:

```
DEFFN artanh(x_r)=LOG((1+x_r)/(1-x_r))/2
```

Area Cotangens hyperbolicus:

```
DEFFN arcoth(x_r)=LOG((x_r+1)/(x_r-1))/2
```

It follows that:

x = @arsinh(@sinh(x)) = @sinh(@arsinh(x))

x = @arcosh(@cosh(x)) = @cosh(@arcosh(x))

x = @artanh(@tanh(x)) = @tanh(@artanh(x))

x = @arcoth(@coth(x)) = @coth(@arcoth(x))

2. Special commands and functions

2.1 The variables

GFA-BASIC and GFA-BASIX provide a range of variables, which - when employed sensibly - result in the optimum usage of memory. For example, it's nonsensical to use a floating point number or a 32 bit integer variable as a loop counter in a loop which only runs from 0 to 200. In terms of memory requirements a byte variable, for example i|, is in this case quite sufficient.

The following variable types are available in GFA-BASIC:

Name	Postfix	Memory requirements	Variable type
Double	#	8 bytes	floating point
Long	%	4 bytes	integer (signed)
Word	&	2 bytes	integer (signed)
Byte	|	1 byte	integer (unsigned)
Boolean	!	1 bit	integer (signed)
String	$	depends on string length	(32767 characters maximum)

2.1.1 Double:

The floating point variable **Double** occupies 8 bytes (64 bits) of memory. The postfix "#" is assumed as default for all variables. This applies only when the variable type is not defined explicitly using **DEFBIT**, **DEFBYT**, **DEFDBL**, **DEFINT**, **DEFLNG**, **DEFWRD**, **DEFSTR** or by attaching a postfix of a specific variable type. The range of available numbers conforms to the IEEE double format and is defined as real numbers between 2.2E-308 and 1.67E+308.

Example:

```
x=123.337 or x#=123.337
```

2.1.2 Long:

The signed integer **Long** occupies 4 bytes (32 bits) of memory. The range of valid numbers for variables with postfix "%" is whole numbers between -2147483648 and +2147483647.

Example:

```
x%=103124
```

2.1.3 Word:

The signed integer **Word** occupies 2 bytes (16 bits) of memory. The range of valid numbers for variables with postfix "&" is whole numbers between -32768 and +32767.

Example:

```
x&=12345
```

2.1.4 Byte:

The unsigned variable type **Byte** occupies 1 byte (8 bits) of memory. The range of valid numbers for variables with postfix "|" is natural numbers between 0 and 255.

Example:

```
x|=209
```

2.1.5 Boolean:

The signed variable type **Boolean** occupies 1 bit of memory. The variables with postfix "!" can only take the values -1 (TRUE) and 0 (FALSE). If a value other than zero is assigned to this variable, it always takes the value of -1, otherwise it defaults to 0. If a logical operation is false this variable takes the value of 0 an, and if the result is true it takes -1.

Example:

```
x!=FALSE              // returns    0
x!=10>20              // returns    0
x!=-100               // returns    -1
```

2.1.6 String:

The memory occupied by strings depends on the number of characters in each string. The spaces are counted as well! The maximum length of variables with postfix "$" is 32767 characters. Since the characters are encoded in ASCII with values between 0 and 255, each character in the string occupies 1 byte of memory.

Every string is defined with the help of a so-called descriptor. A string requires, first of all, 4 bytes of memory for a pointer to a specific memory address. This address contains the address of the descriptor (backtrailer) which occupies 4 bytes of memory. These are followed by 2 bytes (for GFA-BASIC MS-DOS and GFA-BASIC OS/2) or 4 bytes (for GFA-BASIX), which contain the string length. After that follows the string itself. If the number of characters in the string is odd, the string is padded with a filler byte (for GFA-BASIC MS-DOS and GFA-BASIC OS/2) or with as

many bytes as needed (up to 3), so that the total number of characters in the string is divisible by four (for GFA-BASIX).

Example:

x$="Hello GFA"	needs 20 bytes:
4 bytes	for the pointer,
4 bytes	for the address of the descriptor,
2 bytes	for the string length,
9 bytes	for 9 characters, plus 1 byte filler.

The address of any variable can be obtained with functions **VARPTR** or **V:**(= variable pointer) and **ARRPTR** or * (= array-pointer):

Example:

VARPTR(a$) or V:a$	returns the address of the first character in string a$.
{VARPTR(a$)-6} or {V:a$-6}	returns the address of the descriptor for string a$.
INT{VARPTR(a$)-2} or INT{V:a$-2}	returns the length of the string a$.

2.2 Operators

Operators are components of a programming language used to connect and/or compare numeric, logical and string expressions. They can also be referred to as numeric, logical or string operators. The comparisons can also be performed on bit level. These operators are then referred to as bitwise operators.

2.2.1 Floating point operators

The numeric operators in GFA-BASIC are:

`exp1 + exp2 + ... + expn`	addition operator for n => 2 exps.
`+ exp`	sign operator
`exp1 - exp2 - ... - expn`	subtraction operator for n => 2 exps.
`- exp`	sign operator
`exp1 * exp2 * ... * expn`	multiplication operator for n => 2 exps.
`exp1 / exp2 / ... / expn`	division operator for n => 2 exps.
`exp1 ^ exp2 ^ ... ^ expn`	power operator for n => 2 exps.
`exp1 DIV exp2`	whole number division operator
`exp1 MOD exp2`	modulo operator
`MIN(exp1,exp2,...,expn)`	minimum operator for n => 2 exps.
`MAX(exp1,exp2,...,expn)`	maximum operator for n => 2 exps.
`exp, exp1,...,expn:`	floating point variables

The **addition operator** + sums up two or more numeric expressions.

Example:

```
PRINT 3+4+5+6          // displays 18
```

The **subtraction operator** - subtracts two or more numeric expressions.

```
Example:        PRINT 3-4-5-6          // displays -12
```

The **multiplication operator** * gives the product of two or more numeric expressions.

```
Example:        PRINT 1*2*3*4          // displays 24
```

The **division operator** / divides two or more numeric expressions. For more than two exps. the result from the first two exps. is obtained first. This value is then divided by the 3rd exp. This result is then again divided by the 4th exp etc.

```
Example:        PRINT 2/3/4/5          // displays 0.03333333333
                //                        and corresponds to
//                                        ((2/3)/4)/5
```

The **power operator** ^ calculates the power of a number with base exp1 and exponent exp2. For more than two exps the product of exp1^exp2 is calculated first. This value serves as the new base, i.e. (exp1^exp2)^exp3 is calculated next. The resulting value is then again used as the new base etc.

```
Example:        PRINT 5^2^3            // displays 15625
                //                        and corresponds to (5^2)^3
```

The **whole number division operator DIV** calculates the whole part of a division result between two exps. In case of a remainder the number is not rounded off, i.e. TRUNC(exp1/exp2) is calculated.

Example:

```
PRINT 8 DIV 3            // displays 2
```

The **modulo operator MOD** returns the modulo for two exps. It calculates the remainder of exp1/exp2 division and corresponds to exp1-exp2*TRUNC(exp1/exp2).

Example:

```
PRINT 25 MOD 6           // displays 1
```

The application of sign operators + and - to a numeric expression is equivalent to a multiplication of these expressions by +1 or -1, i.e. +exp = (+1)*exp and -exp = (-1)*exp.

The **minimum operator MIN()** returns the minimum value from a list of exps. The individual exps. are separated by commas.

Example:

```
PRINT MIN(2,3.5,-4.2,SQR(25),3)       // displays
//                                        -4.2
```

The **maximum operator MAX()** returns the maximum value from a list of exps. The individual exps. are separated by commas.

Example:

```
PRINT MAX(2,3.5,-4.2,SQR(25),3)       // displays   5
```

2.2.2 Integer operators

The following operators are implemented especially for integer arithmetic (whole number arithmetic):

`ADD(iexp1,iexp2,...iexpn)`	addition operator for n => 2 iexps.
`SUB(iexp1,iexp2)`	subtraction operator for 2 iexps.
`MUL(iexp1,iexp2,...iexpn)`	multiplication operator for n => 2 iexps.
`DIV(iexp1,iexp2)`	division operator for 2 iexps.
`iexp1 \ iexp2`	division operator for 2 iexps.
`iexp1 % iexp2`	modulo operator for 2 iexps.
`IMIN(iexp1,iexp2,...,iexpn)`	minimum operator for n => 2 iexps.
`IMAX(iexp1,iexp2,...,iexpn)`	maximum operator for n => 2 iexps.

The integer operators perform the same operations as the numeric operators but only on whole numbers. However, there are some limitations (**SUB()** and \ for two iexps. only).

Pure integer arithmetic is several times faster than the floating point arithmetic!

2.2.3 Logical operators

The logical operators in GFA-BASIC are:

`erbexp1 && bexp2`	logical	and
`bexp1 || bexp2`	logical	or
`bexp1 ^^ bexp2`	exclusive	or
`!bexp`	negation	

The logical operators are used mainly for conditional branching (see conditional statements). With the exception of negation, they always connect

two expressions and determine the truth of this statement (true or false). The negation negates the expression which follows it.

Example:

```
IF a%>5 && fl1!          // if both a%>5
  //                        and fl1!=-1
  !fl2!                  // then negate fl2!
ELSE IF a%<=5 || a%>0    // if a%<=5
  //                        and/or a%>0
  GOSUB dummy1           // then branch to
  //                        the dummy procedure 1
ELSE IF a%<=5 ^^ a%>3    // if a% is either
  //                        <=5 or >3
  GOSUB dummy2           // then branch to
  //                        the dummy procedure 2
ENDIF
```

It's important here to fully understand the inclusive and exclusive or. The inclusive or (and/or) statement is "true" or "-1" when at least one of the two conditions, or both conditions, are true. For a%=4, the **IF** a%< =5 || a%>0 will cause the branch to be taken, since either of the two conditions (a%< =5 and a%>0) as well as both conditions together are true.

The exclusive or (either/or) is only "true" or "-1" when only one of the two conditions is true. For a%=4, the **IF** a%< =5 ^^ a%>3 will NOT cause the branch to be taken, since both conditions are true here.

The result of a logical comparison is expressed with an actual value. -1 = (logically) "true" and 0 = (logically) "false". Therefore:

AND

Condition 1	Condition 2	Res.
-1	-1	-1
-1	0	0
0	-1	0
0	0	0

OR

Condition 1	Condition 2	Res.
-1	-1	-1
-1	0	-1
0	-1	-1
0	0	0

XOR/either-or

Condition 1	Condition 2	Res.
-1	-1	0
-1	0	0
0	-1	0
0	0	-1

NOT

Condition	Res.
-1	0
0	-1

2.2.4 Bitwise operators

The bit operators in GFA-BASIC are:

```
iexp1 AND iexp2               bitwise and for 2 iexps.
iexp1  &  iexp2               bitwise and for 2 iexps.
iexp1  OR iexp2               bitwise or for 2 iexps.
iexp1  |  iexp2               bitwise or for 2 iexps.
iexp1 XOR iexp2               bitwise exclusive or for 2 iexps.
iexp1 IMP iexp2               bitwise implication for 2 iexps.
iexp1 EQV iexp2               bitwise equivalency for 2 iexps.

AND(iexp1,iexp2,...,iexpn)    bitwise and for n => 2 iexps.
OR(iexp1,iexp2,...,iexpn)     bitwise or for n => 2 iexps.
XOR(iexp1,iexp2,...,iexpn)    bitwise exclusive or for n => 2
                              iexps.

IMP(iexp1,iexp2)              bitwise implication for 2 iexps.
EQV(iexp1,iexp2)              bitwise equivalency for 2 iexps.

NOT iexp                      bitwise negation
  ~ iexp                      bitwise negation, with sign as
                              priority
```

The bitwise operators affect the numeric expressions on bit level. The bits are numbered as follows: 0 is the lowest bit; while (in case of 32-bit integer variables) 31 is the highest bit - also known as the sign bit. If the sign bit is

set the number is negative and it is stored as twos complement (if the bit is clear the number is positive). The value of this (negative) number is then given as (-1)*(the complement of the bit patter) -(1).

The bitwise operators can be used on all integer variables, whereby the **BYTE** and **WORD** variables are first internally expanded to 32 bits. The comparison is then performed on all 32 bits. The result of a bitwise operation is a value produced by combining two different bit patterns.

The **AND operator** sets in the result only the bits which are set in both combined values.

Example:

```
PRINT BIN$(3,4)         // displays   0011
PRINT BIN$(10,4)        // displays   1010
PRINT BIN$(3 AND 10,4)  // displays   0010
```

3 **AND** 10 results in the value 2. This is because the value 3 is coded in binary as 0011 and the value 10 as 1010. If both values are bitwise "and-ed", the result will have a 1 only where both values also have a 1:

0011 **AND** 1010 produces 0010, which is binary for the number 2.

The **inclusive OR operator** sets in the result only the bits which are set in at least one of the two combined values.

Example:

```
PRINT BIN$(3,4)         // displays   0011
PRINT BIN$(10,4)        // displays   1010
PRINT BIN$(3 OR 10,4)   // displays   1011
```

3 **OR** 10 results in the value 11. This is because the value 3 is coded in binary as 0011 and the value 10 as 1010. The inclusive or will therefore set in the result bits 0, 1 and 3.

0011 **OR** 1010 produces 1011, which is binary for the number 11.

The **exclusive or XOR** sets in the result only the bits which are set in one and only one of the two combined values.

Example:

```
PRINT BIN$(3,4)        // displays   0011
PRINT BIN$(10,4)       // displays   1010
PRINT BIN$(3 XOR 10,4) // displays   1001
```

3 **XOR** 10 results in the value 9. This is because the value 3 is coded in binary as 0011 and the value 10 as 1010. The exclusive or will therefore set in the result the bits 0 and 3.

0011 **XOR** 1010 produces 1001, which is binary for the number 9.

•

The **implication operator IMP** combines two expressions based on their bit order.

The result is equivalent to a logical sequence. This means that something is false only when a true statement is followed by a false one. In relation to their binary representation, the combination of two expressions causes a 0 at the corresponding place in the result when the first argument is 1 and the second argument is 0.

Example:

```
PRINT BIN$(3,4)        // displays   0011
PRINT BIN$(10,4)       // displays   1010
PRINT BIN$(3 IMP 10,4) // displays   1110
```

The value 14 is coded in binary as 1110. 3 **IMP** 10 results in the value -2. To understand this all 32 bits must be examined:

```
BIN$(3,32)         =   00000000000000000000000000000011
BIN$(10,32)        =   00000000000000000000000000001010
BIN$(3 IMP 10,32)  =   11111111111111111111111111111110
```

The result of 3 **IMP** 10 is then, as outlined above:

```
(-1)*(00000000000000000000000000000001)-(1)
```

the binary representation of the number -2.

IMP is the only bitwise operator where the order of given arguments is important, since binary representation of the result will contain a 0 only when the bit in the first argument is "true" (1) and the bit at the same position in the second argument is "false" (0). Therefore, 3 **IMP** 10 results in the value -2 (see above). But, 10 **IMP** 3 returns -9:

```
BIN$(10,32)        =    00000000000000000000000000001010
BIN$(3,32)         =    00000000000000000000000000000011
BIN$(10 IMP 3,32)  =    11111111111111111111111111110111
10 IMP 3           =    (-1)*(00000000000000000000000000001000)-(1)
                   =    -9
```

The **equivalence operator EQV** sets in the result only the bits which are the same in both combined values. This is equivalent to **NOT(iexp1 XOR iexp2)**.

Example:

```
PRINT BIN$(3,4)              // displays  0011
PRINT BIN$(10,4)             // displays  1010
PRINT BIN$(3 XOR 10,4)       // displays  1001
PRINT BIN$(3 EQV 10,4)       // displays  0110
PRINT BIN$(NOT(3 XOR 10),4)  // displays  0110
```

3 **EQV** 10 results in the value -10. Again, the results is easier to understand when 32 bit representation is used:

```
BIN$(3,32)          =   00000000000000000000000000000011
BIN$(10,32)         =   00000000000000000000000000001010
BIN$(3 EQV 10,32)   =   11111111111111111111111111110110
3 EQV 10            =   (-1)*(00000000000000000000000000001001)-(1)
                    =   -10
```

2.2.5 Combining strings

The **string operator +** is used to concatenate two or more strings.

Example:

```
a$="today "
b$="and tomorrow"
PRINT a$+b$       //  displays today and tomorrow
```

2.2.6 Relational operators

The following relational operators are implemented in GFA-BASIC:

```
expl =    exp2:    equal operator
aexpl ==  aexp2:   operator for approximate equality
expl >    exp2:    greater than operator
expl =>   exp2:    greater than or equal to operator
expl >=   exp2:    greater than or equal to operator
expl <    exp2:    less than operator
expl <=   exp2:    less than or equal to operator
expl =<   exp2:    less than or equal to operator
expl <>   exp2:    not equal operator
expl ><   exp2:    not equal operator
```

By using relational operators, the numerical, logical and string expressions can be compared to each other. The result of this comparison is always a logical value (-1 for true and 0 for false).

An exception is the operator for approximate equality "= =". It cannot be used to compare strings!

The **equal operator** = compares two expressions to see if they are identical.

Example:

```
PRINT 10=2*5          // displays   -1
PRINT "GFA"="MS"      // displays    0
```

The **operator for approximate equality** == compares two numeric expressions, by comparing the 28 bits of the mantissa of two floating point numbers. The = = operator is relevant only when floating point numbers are used because it can ignore rounding errors

Example:

```
PRINT SINQ(77)==SIN(RAD(77)) // displays     -1
```

The **greater than operator** > compares to expressions and returns -1 (true) when the expression to the left of the operator is greater than the expression on the right.

The strings are compared by comparing the ASCII codes of individual characters. The string "ABC" is handled as numbers 65,66,67. If the comparison reads "ABC">"AAA", the first two characters are compared first. They both have the ASCII code 65. The following two characters are compared next. Since "B" with 66 has the higher ASCII code than "A" with 65, "B" is interpreted as being greater than "A". The comparison of the two strings is them terminated and the value -1 (true) is returned.

A special case of string comparison is when one of the two strings ends before a character is found which is equal, greater than or less than, "AA">"A" for example. This comparison is evaluated as true (-1), since nonexistent characters are interpreted as being "smaller" then the existing characters. "A"+**CHR$**(0)>"A" is therefore also true.

The same applies to greater than or equal to (=>), less than (<), less than or equal to (<=) and not equal (<>) comparisons.

```
expl =>  exp2 returns -1, when expl is greater than or equal to exp2.
expl >=  exp2 returns -1, when expl is greater than or equal to exp2.
expl <   exp2 returns -1, when expl is less than exp2.
expl <=  exp2 returns -1, when expl is less than or equal to exp2.
expl =<  exp2 returns -1, when expl is less than or equal to exp2.
expl <>  exp2 returns -1, when expl and exp2 are different.
expl ><  exp2 returns -1, when expl and exp2 are different.
```

The **assignment operator** = is the most frequently used operator. When the assignment operator is used, the value on the right of the operator is determined first and this value is then assigned to the expression to the left of the operator.

Example:

```
a%=4
b$="Hello GFA"
PRINT a%''b$          // displays   4  Hello GFA
```

2.2.7 Operator hierarchy

When an expression contains several operators they are executed according to a fixed hierarchy. The operators which are higher up in this hierarchy are executed before the ones below.

The hierarchy in GFA-BASIC is as follows:

```
()                          brackets
+                           string concatenation
=, <, >, =>, <=, <>         string comparison
+, -                        sign
^                           powers
*, /                        multiplication, division
DIV, MOD                    whole number division, modulo
+, -                        addition, subtraction
=, <, >, =>, <=, <>         numerical comparison
AND, OR, XOR, IMP, EQV      bitwise comparison
&&,||,^^                    logical comparison
NOT, ~                      bitwise negation
!                           logical negation
```

The operator hierarchy can be changed by using brackets, since the operators within the (innermost) brackets are executed first.

Example:

```
PRINT 2+4*3            // displays   14
PRINT (2+4)*3          // displays   18
PRINT((2+3)*4-5)*2     // displays   30
PRINT 3*2^2            // displays   12
PRINT 8^(2/3)          // displays   4
```

2.3 Type conversion

GFA-BASIC and GFA-BASIX offer a range of functions and commands to enable the conversion of variables from one type to another.

2.3.1 Conversion of numeric into string expressions

The conversion from numeric to string expressions (using the **STR$()** function) is frequently used for formatted output of numeric expressions.

STR$(x)

converts a numeric expression x of any variable type to a string.

Example:

```
PRINT STR$(123.456)    // prints 123.456
```

STR$(x,m)

converts a numeric expression x of any type to a string with the length of m. If m is greater than the number of characters needed to display x, the string is padded with leading spaces. If m is less than the number of characters needed to display x, the string is truncated from the right.

Example:

```
PRINT STR$(123.456,7)  // prints     123.456
PRINT STR$(123.456,5)  // prints     123.4
PRINT STR$(123.456,9)  // prints       123.456
```

STR$(x,m,n)

converts a numeric expression x of any type to a string with the length of m. The parameter n specifies the number of places after the decimal point used for formatting and rounding off. Out of the total length of

m, this reserves n + 1 places (n places for the part after the decimal point and one place for the decimal point itself).

Example:

```
PRINT STR$(123.456,7,3)    // prints  123.456
PRINT STR$(123.456,7,5)    // prints  3.45600
PRINT STR$(123.456,7,2)    // prints   123.46
PRINT STR$(123.456,9,3)    // prints     123.456
```

The reverse function of **STR$()** is **VAL()**!

The **BIN$()**, **OCT$()**, **HEX$()** and **DEC$()** functions convert an integer expression into a string. The expression is converted to binary, octal, hexadecimal or decimal number system respectively.

BIN$(m[,n])

converts the integer expression m to binary display. After the conversion, m is in base 2 number system. In this system only the digits 0 and 1 are used. The optional parameter n specifies how many places (1 to 33) should be used. If n is greater than the number of places needed to display m, the number is padded with leading zeros.

Example:

```
PRINT BIN$(17)          // prints     10001
PRINT BIN$(20+5,6)      // prints     011001
```

OCT$(m[,n]

converts the integer expression m to octal. After the conversion, m is in base 8 number system. In this system only the digits between 0 and 7 are used. The

optional parameter n specifies how many places (1 to 33) should be used. If n is greater than the number of places needed to display m, the number is padded with leading zeros.

Example:

```
PRINT OCT$(17)          // prints    21
PRINT OCT$(25,6)        // prints    000031
```

HEX$(m[,n])

converts the integer expression m to hexadecimal. This is a base 16 number system with digits from 0 to 9 and letters from A to F. The optional parameter n specifies how many places (1 to 33) should be used. If n is greater than the number of places needed to display m, the number is padded with leading zeros.

Example:

```
PRINT HEX$(25)          // prints    19
PRINT HEX$(1001,6)      // prints    0003E9
```

DEC$(m[,n])

converts the integer expression m to decimal. This is a base 10 number system with digits from 0 to 9. The optional parameter n specifies how many places (1 to 33) should be used. If n is greater than the number of places needed to display m, the number is padded with leading zeros.

Example:

```
PRINT DEC$(25)          // prints    25
PRINT DEC$(123,6)       // prints    000123
```

The **CHR$()**, **MKI$()** and **MKL$()** functions convert an 8-, 16- or 32-bit integer expression to one, two or four characters. The **MKS$()** and **MKD$()** functions converts a floating point expression in IEEE single or IEEE double format to four and eight characters.

CHR$(m)

converts the 8-bit integer expression m into a character. **CHR$(m)** returns an ASCII character with the ASCII code m. The value of m can be a natural number in the range between 0 and 255.

Example:

```
PRINT CHR$(65)    //  prints the character A, since
//                    this is what ASCII code 65 is.
```

MKI$(m)

converts the 16-bit integer expression m into two characters. The low byte comes first and the high byte comes second.

Example:

```
PRINT MKI$(24904)     // prints     Ha
```

MKL$(m)

converts the 32-bit integer expression m into four characters. The low byte comes first and the high byte comes last.

Example:

```
PRINT MKL$(1819042120) // prints     Hall
```

MKS$(m)

converts the 32-bit floating point expression m in IEEE single format into four characters. The low byte comes first and the high byte comes last.

Example:

```
PRINT MKS$(100.1)      // prints     100.1
```

MKD$(m)

converts the 64-bit floating point expression m in IEEE double format into eight characters. The low byte comes first and the high byte comes last.

Example:

```
PRINT MKD$(1001.1001)  // prints     1001.1001
```

The reverse function of **CHR$()** is **ASC()**, the reverse functions of **MKI$()**, **MKL$()**, **MKS$()** and **MKD$()** are **CVI()**, **CVL()**, **CVS()** and **CVD()**!

2.3.2 Conversion of strings into numerical expressions

VAL(a$)

converts the string a$ into a number. If **VAL()** encounters a character which cannot be interpreted as a number ("1234a" for example), the conversion is terminated at this point. The number converted up to this point (1234 in the above example) is then returned. If the very first character in the string (excluding -, +, .) cannot be interpreted as a number, **VAL** returns 0.

By using the prefixes &H (HEX), &X (BIN) or &O (OCT), **VAL** can interpret the numbers in hexadecimal, binary and octal. The hexadecimal numbers can also use the prefix $ and the binary numbers can use the prefix %.

Example:

```
PRINT VAL("-.123")      // prints      -0.123
a$=STR$(12345)
PRINT VAL(a$)           // prints      12345
PRINT VAL("&H"+"AF")    // prints      175
PRINT VAL("$AA)         // prints      170
PRINT VAL("%10101011)   // prints      171
```

VAL?(a$)

returns the number of places needed by **VAL(a$)** to convert string a$. **VAL?(a$)** returns 0, is a$ contains no characters which can interpreted as numbers.

Example:

```
PRINT VAL?("12345")       //prints    5
PRINT VAL?("3.00DM)       //prints    4
PRINT VAL?("HALLO GFA")   //prints    0
```

The reverse function of **VAL()** is **STR$()**!

ASC(a$)

converts the first character in string a$ into an 8-bit integer number. **ASC(a$)** returns the ASCII code of the first character in string a$. If a$ is a blank string a 0 is returned.

Example:

```
PRINT ASC("TEST")       // prints 84, since the ASCII
//                         code for T is 84.
```

CVI(a$)

converts the first two characters in string a$ to a 16-bit integer expression. The low byte comes first and the high byte comes second.

Example:

```
PRINT CVI("Hallo GFA") // prints     24904
PRINT CVI(MKI$(24))    // prints     24
```

CVL(a$)

converts the first four characters in string a$ to a 32-bit integer expression. The low byte comes first and the high byte comes last.

Example:

```
PRINT CVL("Hallo")        // prints  1819042120
PRINT CVL(MKL$(123456))   // prints  123456
```

CVS(a$)

converts the first four characters in string a$ to a 32-bit floating point expression in IEEE single format. The low byte comes first and the high byte comes last.

Example:

```
PRINT CVS(MKS$(12.25)) // prints     12.25
```

CVD(a$)

converts the first four characters in string a$ to a 64-bit floating point expression in IEEE double format. The low byte comes first and the high byte comes last.

Example:

```
PRINT CVD(MKD$(1001.1))   // prints  1001.1
```

The functions **ASC** and **CHR$**, **CVI** and **MKI$**, **CVL** and **MKL$**, **CVS** and **MKS$** as well as **CVD** and **MKD$** are the reverse of each other. The examples of where these functions can be used are: the reading of numbers from programs using a different format or place-saving storing of numbers.

2.4 Using strings

Strings can be used in GFA-BASIC as variables and/or as constants.

A constant is used when the string is given explicitly:

Example:

```
ALERT 1, "This is a|string constant",1,"RETURN",a%
```

or

```
PRINT AT(10,10);"This is a string constant"
```

A variable is used when the string is assigned to this variable:

Example:

```
a$="This is a|string variable"
ALERT 1,a$,1,"RETURN",a%
```

or

```
PRINT AT(10,10);a$
```

The string variables are indicated with the $ suffix. Furthermore, all variables with a certain characteristic can be defined as string variables by using the **DEFSTR** command.

DEFSTR s declares all variables, which start with the letter s, as string variables. This assignment can also be achieved by attaching the suffixes #, %, &, | or ! to the corresponding variables.

Example:

```
DEFSTR s
s%=24
ADD s%,18
PRINT s%                // displays 42.
```

Furthermore, GFA-BASIC contains a number of conversion functions, which can change the numeric expressions into character expressions. They are:

CHR$()
MKI$()
MKF$()
MKL$()
MKS$()
MKD$()
STR$()
BIN$()
HEX$()
OCT$()
DEC$().

Additional functions for manipulation of string expressions are:

LEFT$()
RIGHT$()
MID$()
TRIM$()
UPPER$()
UCASE$()
LCASE$()
SPACE$()
MIRROR$()
XLATE$().

The string expression, sexp, is a combination of string constants, string variables and string functions.

Example:
```
a$="GF"
b$=a$+CHR$(65)+"-BASIC"
```

For additional handling of strings GFA-BASIC provides the following functions :

PRED()
SUCC()
INSTR()
RINSTR()

as well as commands

LSET
RSET
MID$.

2.4.1 The String operators

A number of operators can be used on strings. The are

+
=
<>
<
>
<=
=>.

The most important of these operators is the assignment operator =, since it is used for combining strings.

Example:

```
a$="GFA"+"-"+"BASIC"
PRINT a$                 // displays GFA-BASIC
```

For a detailed description of operators refer to the chapter about operators.

The conversion functions have already been described in the chapter about type conversion and should not be explained further.

2.4.2 The Manipulation functions

LEFT$()
RIGHT$()
MID$()
TRIM$()
UPPER$()
UCASE$()
LCASE$()
SPACE$()
MIRROR$()
XLATE$()

The **LEFT$()** function requires two parameters: a string and an integer expression. **LEFT$ (a$,m%)** returns then the first m% characters from the string expression a$. If m% is greater than the number of characters in a$ (both spaces and **CHR$(0)** are also characters!) a$ is returned in full. If m% is less than 1 or if m% is not specified, the first character in a$ is returned.

Example:

```
PRINT LEFT$("Evaluation methods",3)     //     displays Eva
PRINT LEFT$("Evaluation methods",-5)    //     displays E
PRINT LEFT$("Evaluation methods")       //     displays E
```

The **RIGHT$()** function requires the same parameters as **LEFT$()**. **RIGHT$(a$,m%)** returns therefore the last m% characters from the string expression a$.

As with **LEFT$()** the same rule applies: If m% is greater than the number of characters in a$, a$ is returned in full.

If m% is less than 1 or if m% is not specified, **RIGHT$()** returns the last character in a$.

Example:

```
PRINT RIGHT$("Evaluation methods",3)    //     displays ods
PRINT RIGHT$("Evaluation methods",-5)   //     displays s
PRINT RIGHT$("Evaluation methods")      //     displays s
```

The **MID$()** function requires three parameters: one string and two integer expressions. **MID$(a$,p%,m%)** returns m% characters in string a$ starting from position p% (inclusive).

The following applies here: If m% is greater than the number of characters contained in a$, an empty string is returned. If m% is less than 1 or if m% is not specified, the whole string starting from position p% is returned.

Example:

```
PRINT MID$("Evaluation methods",8,3)    //     displays ion
PRINT MID$("Evaluation methods",8,-5)   //     displays ion methods
PRINT MID$("Evaluation methods")        //     displays Evaluation
//                                             methods
```

The **TRIM$()** function requires a string expression as a parameter. **TRIM$**(a$) then removes the leading and trailing spaces in this string.

Example:

```
a$="   Sunday    "
b$="   evening"
c$=TRIM$(a$)+TRIM$(b$)
PRINT c$                    // displays Sundayevening
```

The **UPPER$()** function requires a string expression as a parameter. **UPPER$(a$)** then converts all lowercase letters in a string expression into uppercase.

Example:

```
PRINT UPPER$("Example")    // displays EXAMPLE
```

The **UCASE$()** function is similar to **UPPER$()**, but it only converts the lowercase letters from the American character set (no umlauts or ß).

Example:

```
PRINT UCASE$("Smörebröd") // displays SMöREBRöD
PRINT UPPER$("Smörebröd") // displays SMÖREBRÖD
```

LCASE$() is the reverse function of **UCASE$** and converts, therefore, all uppercase letters in a string to lowercase (except for umlauts and ß).

Example:

```
PRINT LCASE$("ABRACADABRA")  //      displays
//                                   abracadabra
```

The **SPACE$()** function requires as the only parameter an integer expression. **SPACE$(m%)** then creates a string with m% spaces.

Example:

```
PRINT "Hands"+SPACE$(5)+"up" //  displays Hands   up
```

The **MIRROR$()** function requires as the only parameter a string expression. **MIRROR$(a$)** then mirrors the string on the y-axis.

Example:

```
PRINT MIRROR$("Mirror")   // displays rorriM.
```

The **XLATE$** function requires as parameters a string expression and one-dimensional integer array. **XLATE$(a$,m|())** converts the string expression a$ by replacing all of its characters with characters in m|() pointed to by their ASCII codes.

Example:

```
DIM m|(255)
FOR i%=32 TO 255
  m|(i%)=i%
NEXT i%
FOR i%=0 TO 31
  m|(i%)=46
NEXT i%
m|(155)=46
OPEN "i", #1, "a:\test.dat"
PRINT XLATE$(INPUT$(64,#1),m|())
```

Gets 64 bytes from the file TEST.DAT on drive A and sets all control characters and **CHR$**(155) by dots.

The character string functions with the same number of parameters can be interchanged freely.

Example:

```
a$="Heute "
b$="und Morgen"
c$=LEFT$(a$,1)+LEFT$(b$,2)+LEFT$(RIGHT$(b$,3),2)
c$=c$+MID$(b$,7,1)
PRINT c$                 // displays Hunger
```

2.4.3 The Evaluation functions

PRED()
SUCC()
INSTR()
RINSTR()

When used on strings, the **PRED()** function requires a string expression. **PRED(a$)** then returns a character whose ASCII value is one less than the value of the first character in a$.

Example:
```
PRINT PRED("Dent, Arthur")// displays C
```

The **SUCC()** function is similar to **PRED()**. When used on strings, **SUCC()** returns a character whose ASCII value is one greater than the value of the first character in a$.

Example:
```
PRINT SUCC("Dent, Arthur")// displays E
```

The **INSTR()** function requires three parameters, two string and one integer expression. **INSTR(a$,b$,m%)** searches the string expression a$ from the position m% (inclusive) for the substring b$. If m% is not specified, the search is performed from the first character in a$. **INSTR()** then returns the position in a$, from which position b$ is contained within a$. When a$ and b$ are blank strings, **INSTR(a$,b$)** returns the value 1, since the very first character in b$ (space) is equal to a$. If b$ is not found within a$, **INTR(a$,b$)** returns the value 0.

Example: `PRINT INSTR("Coffee beans","ean",5) // displays 9`

The **RINSTR()** function is similar to **INSTR()**. However, **RINSTR** searches through a$ for b$ starting from the right. m% specifies the position from the end of a$.

Example: `PRINT RINSTR("Bohnenkaffee","affe",5)// displays 5`

2.5 The MAT commands

A whole range of **MAT** commands are implemented in GFA-BASIC to facilitate the manipulation of one- and two-dimensional floating point arrays. They are divided into:

1. **System command**: MAT BASE

2. **Generating commands:** MAT CLR, MAT SET, MAT ONE, MAT TRI, MAT DIAG, MAT NEG

3. **Read and write commands:** MAT READ, MAT PRINT, MAT INPUT

4. **Copy and exchange commands:** MAT CPY, MAT XCPY, MAT TRANS

5. **Operating commands:** MAT ADD, MAT SUB, MAT MUL, MAT NORM, MAT DET, MAT QDET, MAT RANK, MAT INV

With the exception of operating commands the versatile **MAT** commands can be used on one- and two-dimensional floating point arrays:

2.5.1 The system command MAT BASE

Due to memory considerations all arrays are initially created with element 0. This is default and is performed by GFA-BASIC with an internal **OPTION BASE 0**. The insertion of line **OPTION BASE 1** in your program means that, from that line on, the array begins with 1. **OPTION BASE** relates to all arrays, regardless of the variable type (integer, floating point, Boolean, string) or the number of dimensions (up to 6).

By using the GFA-BASIC **MAT BASE** command the offset for array indexing of one- and two-dimensional floating point arrays can be changed.

If **OPTION BASE 0** is active, using **MAT BASE 1** will make the indexing of all one- and two-dimensional floating point arrays start with element 1 and 1,1 respectively. **MAT BASE 0** sets this offset back to element (0) and (0,0) respectively. **MAT BASE** has no effect on **OPTION BASE 1** where the indexing always starts with element (1) or (1,1)!

To test whether **OPTION BASE 0** or **OPTION BASE 1** are set in a program you can proceed as follows:

Example:

```
FUNCTION test_option_base
  LOCAL base_%
  ERASE check_option_base&()
  DIM check_option_base&(1)
  base_%=({{*check_option_base&()}}-2) AND (1)
  ERASE check_option_base&()
  RETURN base_%
ENDFUNCTION
```

The function **FUNCTION** test_option_base returns the value 1 when **OPTION BASE 1** is active, or 0 when it's not.

To extend the command **MAT BASE** to any variable type the following subroutine can be used:

Example:

```
PROCEDURE mat_base(o_v%, VAR o_r%)
  o_r%=@test_option_base
  SELECT o_v%
  CASE 0
    o_r%=o_r% OR 0
  CASE 1
    o_r%=o_r% OR 1
  ENDSELECT
RETURN
```

This procedure expects two parameters:

The (call by value) variable o_v% specifies the desired offset for **MAT BASE**.

The (call by reference) variable o_r% specifies the offset for subsequent array indexing.

Only the values 0 and 1 are allowed for "**MAT BASE**". After calling the procedure mat_base, the array indexing must follow the returned offset in o_r%. Within a program this may look as follows:

Example:

```
n%=5
DIM a&(n%)
DATA 1,2,3,4,5,6
base%=@test_option_base
FOR i%=base% TO n%
  READ a&(i%)
  PRINT a&(i%)''        // displays 1  2  3  4  5  6
NEXT i%
PRINT
PRINT
o%=0                    // o% = the variable with
//                         the offset
@mat_base(1,o%)
PRINT "MAT BASE = ";o% // displays MAT BASE = 1
PRINT
FOR i%=o% TO n%
  PRINT a&(i%)''        // displays 2  3  4  5  6
NEXT i%
PRINT
PRINT
@mat_base(0,o%)
PRINT "MAT BASE = ";o% // displays MAT BASE = 0
```

```
PRINT
FOR i%=o% TO n%
  PRINT a&(i%)''        // displays 1  2  3  4  5  6
NEXT i%
```

If **OPTION BASE 1** was set previously, @mat_base(0,o%) has no effect, i.e. the offset o% always has the value 1:

Exampel:

```
OPTION BASE 1
n%=5
DIM a&(n%)
DATA 1,2,3,4,5,6
base%=@test_option_base
FOR i%=base% TO n%
  READ a&(i%)
  PRINT a&(i%)''        // displays 1  2  3  4  5
NEXT i%
PRINT
PRINT
o%=0                    // o% = the variable with the
//                         offset
@mat_base(1,o%)
PRINT "MAT BASE = ";o% // displays MAT BASE = 1
PRINT
FOR i%=o% TO n%
  PRINT a&(i%)''        // displays 1  2  3  4  5
NEXT i%
PRINT
PRINT
@mat_base(0,o%)
PRINT "MAT BASE = ";o% // displays MAT BASE = 1
PRINT
FOR i%=o% TO n%
  PRINT a&(i%)''        // displays 1  2  3  4  5
NEXT i%
```

However, you should bear in mind that for **OPTION BASE 0** the array indexing always starts with element 0, i.e. **DIM** a(5) creates a floating point array for 6 (0, 1, 2, 3, 4, 5) entries. The internal GFA-BASIC **MAT BASE** as well as the "**MAT BASE**" in the above described procedure have no effect on this. **MAT BASE** influences only the indexing of array variables!

The setting defined with GFA-BASIC command **MAT BASE** applies only to commands

MAT READ
MAT PRINT
MAT CPY
MAT XCPY
MAT ADD
MAT SUB
MAT MUL.

The default is **MAT BASE 1**.

2.5.2 The generating commands

MAT CLR
MAT SET
MAT NEG
MAT ONE
MAT DIAG
MAT TRI

The **MAT CLR** command sets all elements in one- or two-dimensional floating point arrays to 0. **MAT CLR a()** corresponds to an **ARRAYFILL a(),0. ARRAYFILL** is, therefore, a very versatile command, since by using this command arrays of any numeric variable type or dimension (up to 6) can be initialised. The examples of **ARRAYFILL** usage are

```
ARRAYFILL a()=0
ARRAYFILL a%()=10
ARRAYFILL a&()=4
ARRAYFILL a!()=TRUE
```

The **MAT SET** command sets all elements in a one- and two-dimensional floating point array to the given value. **MAT SET** a() = -2 corresponds, therefore, to an **ARRAYFILL** a(),-2.

The **MAT NEG** command reverses the signs of all elements in a one- or two-dimensional floating point array.

Example:

```
DIM a(5)
DATA 1,3,-2,-1,-1
MAT READ a()
MAT PRINT a()          // displays   1,3,-2,-1,-1
PRINT
MAT NEG a()
MAT PRINT a()          // displays   -1,-3,2,1,1
```

The **MAT ONE, MAT DIAG** and **MAT TRI** commands can only be used on two-dimensional floating point arrays with the same number of rows and columns!

To better understand the following explanation here's a small course in the terminology of linear algebra first:

The numeric arrays of any dimension are called matrices in linear algebra. The most frequently used matrices have one or two dimensions, i.e. they are one- or two-dimensional arrays.

If a numeric array of m * n expressions is arranged in a square with m rows and n columns,

```
for example  a(1,1) a(1,2) ... a(1,n)
             a(2,1) a(2,2) ... a(2,n)
             .      .      ... .
             a(m,1) a(m,2) ... a(m,n)
```

this is then a matrix of (m,n) type. The **m*n** expressions a(i,j) are called the elements of matrix a(). The position of an element within the matrix is described with a twin index, where the first index is the row and the second index is the column where the element is located. The numbering of rows runs from top to bottom and the numbering of column from left to right. (The elements of a matrix are, as a rule, numbers).

The matrices with only one dimension (i.e. with only one row or only one column) are called vectors or row/column matrices.

If a matrix a() contains the same number of rows and columns (for example **DIM** a(5,5)), it's referred to as a quadratic (square) matrix of n-th order.

Example:

```
1 2 3 4 5
2 3 4 5 6
4 5 6 7 8
2 3 4 5 6
1 2 4 5 6
```

In the above example a() is therefore a square matrix of the 5th order.

The elements in a square matrix with the same row and column index (in the above example the elements a(1,1), a(2,2), a(3,3), a(4,4) and a(5,5))

create the main diagonal of a square matrix. In geometric terms a square matrix can be seen as a square. If a line is drawn from the upper left to the lower right corner, all elements of the main diagonal lie on this line. The main diagonal in the above matrix is defined by (1,1)=1, (2,2)=3, (3,3)=6, (4,4)=5 and (5,5)=6.

The general rule is that the elements a(i,i); i = **MIN**(number of rows and columns) are the elements of a main diagonal.

The sum of all elements in the main diagonal of a matrix is called a track. The track of the matrix in the above example = 21.

If all elements in a square matrix - with the exception of the main diagonal - are 0, it's referred to as a diagonal matrix.

```
Example:        1 0 0 0 0
                0 3 0 0 0
                0 0 6 0 0
                0 0 0 5 0
                0 0 0 0 1
```

If all elements of a square matrix below the main diagonal, i.e. all elements whose row index is bigger than their column index, are 0, such a matrix is known as an upper triangular matrix. The reverse, a square matrix all of whose elements above the main diagonal (bigger column than row index) are 0, is described as a lower triangular matrix. Each diagonal matrix is therefore simultaneously an upper and a lower triangular matrix.

Example:

```
upper triangular matrix       lower triangular matrix
1 2 3 4 5                     1 0 0 0 0
0 3 2 5 6                     2 3 0 0 0
0 0 4 7 9                     3 2 4 0 0
0 0 0 2 1                     4 5 7 2 0
0 0 0 0 5                     5 6 9 1 5
```

A square matrix, all elements of which follow the formula a(i,j) = a(j,i), for i = 1,... number of rows and j = 1,... number of columns, is called a symmetrical matrix. Each diagonal matrix is therefore simultaneously a symmetrical matrix.

Example:

```
1 2 3 4 5
2 7 6 8 9
3 6 4 2 1
4 8 2 7 5
5 9 1 5 3
```

A special form of the diagonal matrix is a uniform matrix. The uniform matrix is defined as a diagonal matrix, whose diagonal elements are all equal to 1.

Example:

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

The **MAT ONE** command creates a uniform matrix out of the given square matrix.

Example:

```
DIM a(3,3)
MAT ONE a()
MAT PRINT a()

displays
1, 0, 0
0, 1, 0
0, 0, 1
```

The **MAT DIAG** command creates a diagonal matrix with freely selectable value for the main diagonal elements, out the a given square matrix:

Example:

```
DIM a(3,3)
MAT DIAG a(),4
MAT PRINT a()

displays
4, 0, 0
0, 4, 0
0, 0, 4
```

The **MAT TRI** command creates a lower (parameter = 0) or an upper (parameter = 1) triangular matrix out of the given square matrix.

Example:

```
DIM a(3,3)
MAT SET a()=5
MAT TRI a(),0
MAT PRINT a()

displays
5, 0, 0
5, 5, 0
5, 5, 5

DIM a(3,3)
MAT SET a()=5
MAT TRI a(),1
MAT PRINT a()

displays
5, 5, 5
0, 5, 5
0, 0, 5
```

2.5.3 The read and write commands

MAT READ
MAT PRINT
MAT INPUT

MAT READ a() reads in the values of a one- or two-dimensional floating point array from the **DATA** lines.

Example:

```
DIM a(4,3)
DATA 1,2,3,4,5,6,7,8,9,9,8,7
MAT READ a()
MAT PRINT a()

displays
1, 2, 3
4, 5, 6
7, 8, 9
9, 8, 7

DIM a(12)
DATA 1,2,3,4,5,6,7,8,9,9,8,7
MAT READ a()
MAT PRINT a()


displays
1,2,3,4,5,6,7,8,9,9,8,7
```

MAT PRINT prints a one- or two-dimensional floating point array. The output can be redirected - like with **PRINT** - to a file channel. In addition, the output can be formatted like with **STR$(x,g,n)**.

Example:

```
DIM a(3,3)
DATA 1,2.3333,3
DATA 7,5.25873,9.376
DATA 3.23,7.2, 8.999
MAT READ a()
MAT PRINT a()

displays
1,2.3333,3
7,5.25873,9.376
3.23,7.2,8.999

MAT PRINT a(),7,3

displays
1.000,2.333,3.000
7.000,5.259,9.376
3.230,7.200,8.999
```

During output to a file, one-dimensional arrays are laid out on a line with individual elements separated by commas. For two-dimensional arrays each row is followed by a line feed. The file channel must previously have been opened with **OPEN** "o",#...

Example:

```
DIM a(3,3)
DATA 1,2.3333,3
DATA 7,5.25873,9.376
DATA 3.23,7.2, 8.999
MAT READ a()
OPEN "o",#1,"A:\TEST.DAT"
MAT PRINT #1,a()
CLOSE #1
```

MAT INPUT reads a one- or two-dimensional floating point array in ASCII from a file (the format of which is the reverse of **MAT PRINT**, the commas and line feeds are used as in **INPUT #**).

Example:

```
DIM a(3,3)
DATA 1,2.3333,3
DATA 7,5.25873,9.376
DATA 3.23,7.2, 8.999
MAT READ a()
OPEN "o",#1,"A:\TEST.DAT"
MAT PRINT #1,a()
CLOSE #1
MAT CLR a()
MAT PRINT a()

displays
0,0,0
0,0,0
0,0,0

OPEN "i",#2,"A:\TEST.DAT"
MAT INPUT #2,a()
CLOSE #2
MAT PRINT a(),7,3

displays
1.000,2.333,3.000
7.000,5.259,9.376
3.230,7.200,8.999
```

2.5.4 The copy and exchange commands

MAT CPY
MAT XCPY
MAT TRANS

MAT CPY copies a square segment in a one- or two-dimensional floating point array from a particular row and column offset to another row and column offset in a second one- or two-dimensional floating point array.

MAT CPY a([i,j])=b([k,l])[,h,w] copies a square segment with h rows and w elements, from row/column offset k,l in floating point array b() to i,j row/column offset in the array a().

Example:

```
DIM a(3,5),b(6,6)
MAT SET a()= -1
MAT SET b()=5
MAT CPY a(2,2)=b(3,4),3,3
MAT PRINT a(),5,2

displays
-1.00,-1.00,-1.00,-1.00,-1.00
-1.00, 5.00, 5.00, 5.00,-1.00
-1.00, 5.00, 5.00, 5.00,-1.00
```

If **MAT CPY** is used on a one-dimensional floating point array, j and l are ignored. The **MAT CPY** following a **DIM a(n),b(m)** interprets a() and b() as row vectors, i.e. as matrices of (1,n) and (1,m) type, without them having to be dimensioned explicitly.

Example:

```
DIM a(10),b(10)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(3,1)=b(4,1),5,2
MAT PRINT a()
```

```
displays
1,1,2,2,2,2,2,1,1,1

DIM a(10),b(10)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(1,3)=b(1,4),5,2
MAT PRINT a()

displays
2,2,2,2,2,1,1,1,1,1
```

Should **MAT CPY** be used on column vectors, a() and b() must be explicitly defined as two-dimensional floating point arrays of (n,1) and (m,1) type.

Example:

```
DIM a(10,1),b(10,1)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(3,1)=b(4,1),5,2
MAT PRINT a(),4,2

displays
1.00
1.00
2.00
2.00
2.00
2.00
2.00
1.00
1.00
1.00
```

When the parameters h and w are specified for **MAT CPY** the following applies to copying one-dimensional floating point arrays:

for **w** => **1** only the parameter h is observed.
for **w** = **0** no copying takes place.
for **h** = **0** no copying takes place.

Example:

```
DIM a(10),b(10)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(3,1)=b(4,1),5,4 // w=4; =>1
MAT PRINT a()

displays
1,1,2,2,2,2,2,1,1,1
```

```
DIM a(10),b(10)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(3,1)=b(4,1),5,0 // w=0
MAT PRINT a()

displays
1,1,1,1,1,1,1,1,1,1
```

```
DIM a(10),b(10,1)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(3,1)=b(4,1),0,5 // h=0
MAT PRINT a()

displays
1,1,1,1,1,1,1,1,1,1
```

By leaving out some of the indices i,j or k,l or the given height (h) and width (w) a **MAT CPY** can result in the following special cases:

MAT CPY a() = b() copies to a(), all elements from the floating point array b() for which there are identical indices in floating point array a().

Example:

```
DIM a(3,5),b(6,6)
MAT SET a()=1
MAT SET b()=2
MAT CPY a()=b()
MAT PRINT a()

displays
2,2,2,2,2
2,2,2,2,2
2,2,2,2,2
```

MAT CPY a(i,j) = b() copies all elements in floating point array b(), from row/column offset defined with **MAT BASE**, to row/column offset defined with i and j in floating point array a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the arrays, the number of rows (h) and the number of elements per row (w).

Example:

```
DIM a(3,5),b(6,6)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(2,2)=b()
MAT PRINT a()

displays
1,1,1,1,1
1,2,2,2,2
1,2,2,2,2
```

MAT CPY a()=b(k,l) copies all elements in floating point array b(), from row/column offset defined with k and l, to row/column offset defined with **MAT BASE** in floating point array a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the arrays, the number of rows (h) and the number of elements per row (w).

Example:

```
DIM a(3,5),b(6,6)
MAT SET a()=1
MAT SET b()=2
MAT CPY a()=b(4,4)
MAT PRINT a()

displays
2,2,2,1,1
2,2,2,1,1
2,2,2,1,1
```

MAT CPY a(i,j)=b(k,l) copies all elements in floating point array b(), from row/column offset defined with k and l, to row/column offset defined with i and j in floating point array a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the arrays, the number of rows (h) and the number of elements per row (w).

Example:

```
DIM a(3,5),b(6,6)
MAT SET a()=1
MAT SET b()=2
MAT CPY a(2,2)=b(4,4)
MAT PRINT a()
```

```
displays
1,1,1,1,1
1,2,2,2,1
1,2,2,2,1
```

MAT CPY a()=b(),h,w copies h rows and w elements in floating point array b(), from row/column offset defined with **MAT BASE**, to row/column offset defined with **MAT BASE** in floating point array a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the array, the number of rows (h) and the number of elements per row (w).

Example:

```
DIM a(3,5),b(6,6)
MAT SET a()=1
MAT SET b()=5
MAT CPY a()=b(),3,3
MAT PRINT a()

displays
2,2,2,1,1
2,2,2,1,1
2,2,2,1,1
```

MAT XCPY works in principle like **MAT CPY**. However, during the copying of the source array into the target array the rows and columns are swapped. The source array remains unchanged. Only the elements for which indices exist in both source and target arrays are copied.

MAT XCPY a([i,j])=b([k,l])[,h,w] copies a rectangular segment with h rows and w elements, from row/column offset defined with l and k in floating point array b(), to row/column offset defined with i and j in floating point array a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the arrays, the number of rows (h) and the number of elements per row (w). The matrix b(), or

the relevant part of it, are internally transposed before copying, that is to say the rows and column are swapped. This change affects only the copy and not the matrix b() itself.

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%= 1 TO 4
  FOR j%=1 TO 4
    b(i%,j%)=SUCC(j%)
  NEXT j%
NEXT i%
MAT XCPY a(2,2)=b(2,2),3,3
MAT PRINT b()
PRINT "-------"
MAT PRINT a()
```

```
displays
2,3,4,5
2,3,4,5
2,3,4,5
2,3,4,5
-------
1,1,1
1,3,3
1,4,4
1,5,5
1,1,1
```

If **MAT XCPY** is used on a one-dimensional floating point array, j and l are ignored. The **MAT CPY** following a **DIM a(n),b(m)** interprets a() and b() as row vectors, i.e. as matrices of (1,n) and (1,m) type, without them having to be dimensioned explicitly. Since **MAT XCPY** performs a row/column swap before copying, in case of one-dimensional floating point array **MAT XCPY** will only copy one element from the source to the target array.

Example:

```
DIM a(10),b(10)
MAT SET a()=1
MAT SET b()=2
MAT XCPY a(3,1)=b(4,1),5,4
MAT PRINT a()

displays
1,1,2,1,1,1,1,1,1,1
```

As was the case with **MAT CPY** - by partially loosing the indices i,j or k,l or the given height (h) and width (w) the following special cases may occur:

```
MAT XCPY a()=b()
MAT XCPY a([i,j])=b()
MAT XCPY a()=b([k,l])
MAT XCPY a()=b(),w,h
```

These act the same as the corresponding **MAT CPY** commands, except for the transposition of relevant areas of matrix b() before copying to matrix a().

MAT TRANS copies a two-dimensional floating point array to another two-dimensional floating point array. The rows and column are thereby exchanged. **MAT TRANS a()=b()** corresponds, therefore, to a **MAT XCPY a()=b()**. Because of the row/column swap, care must be taken when using **MAT TRANS** that the target matrix has the same number of rows as the source matrix has columns. In addition, the target matrix must have the same number of columns as the source matrix has rows (for example **DIM** a(5,3),b(3,5)).

Example:

```
DIM a(4,3),b(3,4)
MAT SET a()=1
FOR i%=1 TO 3
  FOR j%=1 TO 4
    b(i%,j%)=j%
  NEXT j%
NEXT i%
MAT TRANS a()=b()
MAT PRINT b()
PRINT "-------"
MAT PRINT a()

displays
1,2,3,4
1,2,3,4
1,2,3,4
-------
1,1,1
2,2,2
3,3,3
4,4,4
```

If a floating point array a() has the same number of rows and columns, the **MAT TRANS a()** command can be used. It swaps the rows and columns in a(), and writes the modified array back to a(). The original array a() is thereby lost. (However, it can be restored by performing **MAT TRANS a()** again.)

Example:

```
DIM a(4,4)
FOR i%=1 TO 4
  FOR j%=1 TO 4
    a(i%,j%)=j%
  NEXT j%
NEXT i%
```

```
MAT PRINT a()
PRINT "-------"
MAT TRANS a()
MAT PRINT a()

displays
1,2,3,4
1,2,3,4
1,2,3,4
1,2,3,4
-------
1,1,1,1
2,2,2,2
3,3,3,3
4,4,4,4
```

2.5.5 The operating commands

MAT ADD
MAT SUB
MAT MUL
MAT NORM
MAT DET
MAT QDET
MAT RANK
MAT INV

In contrast to already described **MAT** commands, the operating commands perform linear algebra matrix operations.

MAT ADD adds one- or two-dimensional floating point arrays according to the rules of matrix addition.

The rules of matrix addition:

Only the matrices of the same order, i.e. with the same number of rows and columns, can be added. The addition of individual elements is performed as follows:

```
a(i,j)=b(i,j)+c(i,j); i = 1,..., number of rows; j = 1,...,number of
                                                           columns
```

A single value, called a scalar, can be added to any matrix. This scalar is added to each value in the matrix.

```
a(i,j)=a(i,j)+x; i = 1,..., number of rows; j = 1,...,number of columns
```

MAT ADD a() = b() + c() presupposes that the floating point arrays a(),b() and c() have the same number of dimensions (**DIM** a(2,3),b(2,3),c(2,3) for example). For every i = 1,...,number of rows and j = 1,...,number of columns, the element c(i,j) is added to the element b(i,j) and the result is written to a(i,j).

Example:

```
DIM a(3,5),b(3,5),c(3,5)
MAT SET b()=3
MAT SET c()=4
MAT PRINT b()
PRINT STRING$(10,"-")
MAT PRINT c()
PRINT STRING$(10,"-")
MAT ADD a()=b()+c()
MAT PRINT a()

displays
3,3,3,3,3
3,3,3,3,3
3,3,3,3,3
----------
4,4,4,4,4
```

```
4,4,4,4,4
4,4,4,4,4
----------
7,7,7,7,7
7,7,7,7,7
7,7,7,7,7
```

MAT ADD a(),b() presupposes that the floating point arrays a() and b() have the same number of dimensions (**DIM** a(2,3),b(2,3) for example). For every i = 1,...,number of rows and j = 1,...,number of columns the element b(i,j) is added to element a(i,j) and the result is written to a(i,j). The original element a(i,j) is thereby lost.

Example:

```
DIM a(3,5),b(3,5)
MAT SET a()=1
MAT SET b()=3
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT ADD a(),b()
MAT PRINT a()

displays
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
---------
3,3,3,3,3
3,3,3,3,3
3,3,3,3,3
---------
4,4,4,4,4
4,4,4,4,4
4,4,4,4,4
```

MAT ADD a(),x adds the scalar x to the element a(i,j), for every i = 1,...,number or rows and j = 1,...,number of columns, and writes the result to a(i,j). The original element a(i,j) is thereby lost.

Example:

```
DIM a(3,5)
MAT SET a()=1
MAT PRINT a()
PRINT STRING$(10,"-")
MAT ADD a(),5
MAT PRINT a()

displays
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
---------
6,6,6,6,6
6,6,6,6,6
6,6,6,6,6
```

MAT SUB subtracts one- or two-dimensional floating point arrays according to the rules of matrix subtraction.

The rules of matrix subtraction:

Only the matrices of the same order, i.e. with the same number of rows and columns, can be subtracted. The subtraction of individual elements is performed as follows:

```
a(i,j)=b(i,j)-c(i,j); i = 1,..., number of rows; j = 1,...,number of
                                                        columns
```

A single value, the scalar, can be subtracted from any matrix. This scalar is subtracted from each value in the matrix.

```
a(i,j)=a(i,j)-x; i = 1,..., number of rows; j = 1,...,number of columns
```

MAT SUB **a()=b()-c()** presupposes that the floating point arrays a(),b() and c() have the same number of dimensions (**DIM** a(2,3),b(2,3),c(2,3) for example). For every i = 1,...,number of rows and j = 1,...,number of columns the element c(i,j) is subtracted from the element b(i,j) and the result is written to a(i,j).

Example:

```
DIM a(3,5),b(3,5),c(3,5)
MAT SET b()=6
MAT SET c()=4
MAT PRINT b()
PRINT STRING$(10,"-")
MAT PRINT c()
PRINT STRING$(10,"-")
MAT SUB a()=b()-c()
MAT PRINT a()

displays
6,6,6,6,6
6,6,6,6,6
6,6,6,6,6
---------
4,4,4,4,4
4,4,4,4,4
4,4,4,4,4
---------
2,2,2,2,2
2,2,2,2,2
2,2,2,2,2
```

MAT SUB a(),b() assumes that the floating point arrays a() and b() have the same number of dimensions (**DIM** a(2,3),b(2,3) for example). For every i = 1,...,number of rows and j = 1,...,number of columns the element b(i,j) is subtracted from the element a(i,j) and the result is written to a(i,j). The original element a(i,j) is thereby lost.

Example:

```
DIM a(3,5),b(3,5)
MAT SET a()=3
MAT SET b()=1
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT SUB a(),b()
MAT PRINT a()

displays
3,3,3,3,3
3,3,3,3,3
3,3,3,3,3
---------
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
---------
2,2,2,2,2
2,2,2,2,2
2,2,2,2,2
```

MAT SUB a(),x subtracts the scalar x from the element a(i,j), for every i = 1,...,number or rows and j = 1,...,number of columns, and writes the result to a(i,j). The original element a(i,j) is thereby lost.

Example:

```
DIM a(3,5)
MAT SET a()=5
MAT PRINT a()
PRINT STRING$(10,"-")
MAT SUB a(),2
MAT PRINT a()

displays
5,5,5,5,5
5,5,5,5,5
5,5,5,5,5
---------
3,3,3,3,3
3,3,3,3,3
3,3,3,3,3
```

MAT MUL multiplies one- or two-dimensional floating point arrays according to the rules of matrix multiplication.

The rules of matrix multiplication:

In order to get a product of two matrices, the matrix on the left must have the same number of columns as the matrix on the right has rows.

The matrices are multiplied based on the formula 'row times column'. One gets the element a(i,j) of the resulting matrix, when one multiplies the elements on the i-th row from the matrix on the left with the elements in the j-th column from the matrix on the right and adds up the individual products:

Let's assume that m contains the number of rows and n the number of columns in matrix a().

In order to get the product of matrices b() and c(), m must then also be the number of rows in matrix b() and n the number of columns in matrix c(). Furthermore, o must be the number of columns in matrix b() and at the same time the number of rows in matrix c() (**DIM** a(5,4), b(5,3), c(3,4) for example).

```
a(i,j) = b(i,1)*c(1,j) + b(i,2)*c(2,j), +, ... +b(i,o)*c(o,j)
for i  =  1,...,m and j = 1,...,n
```

Out of this general rule four special cases can be deduced:

1. The left matrix is a matrix of (m,n) type, the right is a column vector of (n,1) type. The result of this product is then a column vector of (m,1) type.

2. The left matrix is a row vector of (1,m) type, the right is a matrix of (m,n) type. The result of this product is then a row vector of (1,n) type.

3. The left matrix is a column vector of (m,1) type, the right is a row vector of (1,n) type. The result of this product is then a matrix of (m,n) type. This matrix is called a dyadic product of two vectors.

4. The left matrix is a row vector of (1,m) type, the right a column vector of (m,1) type. The result of this product is then a single value. This value is called a scalar product of two vectors.

Any matrix can be multiplied with a single value - the scalar. This results in each element of the matrix being multiplied with the scalar:

```
a(i,j)=a(i,j)*x; i = 1,...,m; j = 1,...,n
```

MAT MUL a()=b()*c() presupposes that the floating point arrays a(),b() and c() are dimensioned according to the above definition (**DIM** a(2,3),b(2,4),c(4,3) for example). For k = 1,...,o the elements b(i,k) and c(k,j) are multiplied and added. This sum always yields the elements a(i,j), i = 1,...,m; j = 1,...,n.

Example: **matrix times matrix:**

```
DIM a(2,2),b(2,3),c(3,2)
MAT SET b()=1
DATA 1,2,-3,4,5,-1
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(17,"-")
MAT PRINT c(),5,1
PRINT STRING$(17,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1

displays
 1.0,  1.0,  1.0
 1.0,  1.0,  1.0
----------------
 1.0,  2.0
-3.0,  4.0
 5.0, -1.0
----------------
 3.0,  5.0
 3.0,  5.0
```

Example: **matrix times vector**

```
DIM a(2),b(2,3),c(3)
MAT SET c()=1
DATA 1,2,-3,4,5,-1
MAT READ b()
MAT PRINT b(),5,1
PRINT STRING$(17,"-")
```

```
MAT PRINT c(),5,1
PRINT STRING$(17,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
```

displays

```
1.0,  2.0, -3.0
4.0,  5.0, -1.0
---------------
1.0,  1.0,  1.0
---------------
0.0,  8.0
```

Example: **vector times matrix**

```
DIM a(2),b(3),c(3,2)
MAT SET b()=1
DATA 1,2,-3,4,5,-1
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(17,"-")
MAT PRINT c(),5,1
PRINT STRING$(17,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
```

displays

```
 1.0,  1.0,  1.0
----------------
 1.0,  2.0
-3.0,  4.0
 5.0, -1.0
----------------
 3.0,  5.0
```

Example: **vector times vector (dyadic product):**

```
DIM a(3,3),b(3),c(3)
DATA 1,2,-3,4,5,-1
MAT READ b()
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(17,"-")
MAT PRINT c(),5,1
PRINT STRING$(17,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
```

displays

```
  1.0,   2.0, -3.0
-----------------
  4.0,   5.0, -1.0
-----------------
  4.0,   5.0, -1.0
  8.0,  10.0, -2.0
-12.0, -15.0,  3.0
```

Example: vector times vector (scalar product):

MAT MUL x=a()*b() presupposes that the floating point arrays a() and b() are dimensioned according to the above definition. For this special GFA-BASIC command it is sufficient to define both arrays as one-dimensional floating point arrays with the same number of elements (**DIM** a(3),b(3) for example). For k = 1,...,o the elements a(k) and b(k) multiplied and added up. This sum yields the value x.

Example:

```
DIM a(3),b(3)
DATA 1,2,-3,4,5,-1
MAT READ a()
MAT READ b()
MAT PRINT a(),5,1
PRINT STRING$(17,"-")
```

```
MAT PRINT b(),5,1
PRINT STRING$(17,"-")
MAT MUL x=a()*b()
PRINT x

displays
 1.0,  2.0, -3.0
----------------
 4.0,  5.0, -1.0
----------------
17.0
```

If more then two matrices are to be multiplied the calculation is performed from beginning to end or vice versa. The first two (or the last two) matrices are multiplied with each other first. The resulting matrix is then multiplied with the third (or with the third last) matrix and so on. One special, although frequently used case of a scalar product occurs when a vector is multiplied with a matrix and the resulting vector is multiplied with a vector again. The GFA-BASIC command **MAT MUL x=a()*b()*c()** is used in this case.

Example:

```
DIM a(2),b(2,3),c(3)
DATA 1,2,-3,4,5
MAT READ a()
MAT READ c()
MAT SET b()=1
MAT PRINT a(),5,1
PRINT STRING$(17,"-")
MAT PRINT b(),5,1
PRINT STRING$(17,"-")
MAT PRINT c(),5,1
PRINT STRING$(17,"-")
MAT MUL x=a()*b()*c()
PRINT x
```

```
displays
 1.0,  2.0
 ---------------
 1.0,  1.0,  1.0
 1.0,  1.0,  1.0
 ---------------
-3.0,  4.0,  5.0
 ---------------
18.0
```

An interesting application of matrix multiplication is given in the following example:

Consider the next five railway stations in united Germany:

1. Hamburg
2. Düsseldorf
3. Berlin
4. Leipzig
5. Munich

Define now if there is a direct connection between stations A and B as follows. If there is, mark it with a 1, if not with a 0. In the above example you would thereby get the following (fictional) matrix v():

	Hamburg	**Düsseldorf**	**Berlin**	**Leipzig**	**Munich**
Hamburg	0	1	0	0	0
Düsseldorf	0	0	1	1	1
Berlin	1	1	0	1	0
Leipzig	0	0	1	0	0
Munich	0	1	0	1	0

The 1 on the row "Hamburg" means that there is a direct connection from Hamburg to Düsseldorf, while both 1s in the "Berlin" column mean that there is a direct connection to Düsseldorf and Leipzig. The rows with the highest sums are "Berlin" and "Düsseldorf" rows, which means, that Berlin and Düsseldorf have the highest number of direct connections to other stations. The columns with the highest sums are Düsseldorf and Leipzig, which means, that both of these stations offer the highest number of connections to other stations.

If a matrix product v()*v() = v()^2 is performed, it results in:

	Hamburg	Düsseldorf	Berlin	Leipzig	Munich
Hamburg	0	0	1	1	1
Düsseldorf	1	2	1	2	0
Berlin	0	1	2	1	1
Leipzig	1	1	0	1	0
Munich	0	0	2	1	1

The main diagonal of this matrix indicates how many direct return connections exist between the given stations. The 2 on the row "Düsseldorf" and column "Düsseldorf" indicates, for example, that a return connection exists between Düsseldorf - Berlin, Berlin - Düsseldorf and Düsseldorf - Munich, Munich - Düsseldorf. The elements outside the main diagonal give the number of connections via another station. The 2 on the row "Munich" and in the column "Berlin" indicate that one can go from Munich to Berlin either via Düsseldorf or via Leipzig. The 2 on the row "Düsseldorf" and column "Leipzig" indicates that one can reach Leipzig from Düsseldorf either via Berlin or via Munich.

If a matrix product v()*v()' is performed, where v()' is the transposed v(), it results in:

	Hamburg	Düsseldorf	Berlin	Leipzig	Munich
Hamburg	1	0	1	0	1
Düsseldorf	0	3	1	1	1
Berlin	1	1	3	0	2
Leipzig	0	1	0	1	0
Munich	1	1	2	0	2

The main diagonal of this matrix corresponds to the row sums of the output matrix v(), i.e. it gives the number of direct connections from a station to other stations. The elements outside the main diagonal show how many direct connections there are from a "row" station to other stations, as well as from "column" stations. So a 2 in row "Munich" and column "Berlin" means that a direct connection to stations Düsseldorf and Leipzig exists from both Munich as well as Berlin.

If a matrix product v'()*v() is performed, where v()' is the transposed v(), it results in:

	Hamburg	Düsseldorf	Berlin	Leipzig	Munich
Hamburg	1	1	0	1	0
Düsseldorf	1	3	0	2	0
Berlin	0	0	2	1	1
Leipzig	1	2	1	3	1
Munich	0	0	1	1	1

The main diagonal of this matrix corresponds to the column sums of the output matrix v(), i.e. it gives the number of direct connections from other stations to the selected station. The elements outside the main diagonal show how many direct connections there are from other stations to "row" stations, as well as to "column" stations. So a 2 in row "Düsseldorf" and column "Leipzig" means that a both Berlin as well as Munich have a direct connection to Düsseldorf and Leipzig.

Similar matrix multiplications are also used for demographic research, for example election analysis or insurance calculations to ascertain the risk factor.

The vectors are defined by their norm or length. This is the positive square root of the scalar product of the vector with itself. In other words, the length of a vector is given by the positive square root of the sum of squares of the elements in the vector:

Let's assume that n contains the number of elements in vector a(). To get no, the norm of a(), one would use the formula no=**SQR**(a(1)^2+a(2)^2+...,+a(n)^2).

If every element of a vector is divided by its norm, the vector becomes a normalised vector. The length of a normalised vector is equal to 1:

```
1 = SQR((a(1)/no)^2+(a(2)/no)^2+,....,+(a(n)/no)^2)
```

MAT NORM a(),0 and **MAT NORM a(),1** create a normalised vector out of the vector a(). The original vector a() is thereby lost. If a() is a matrix, **MAT NORM a(),0** creates a matrix whose rows are normalised and **MAT NORM a(),1** creates a matrix whose columns are normalised.

Example:

```
n%=8
DIM a(n%,n%),b(n%,n%),v(n%)
DATA 1,2,3,4,5,6,7,8
DATA 3.2,4,-5,2.4,5.1,6.2,7.2,8.1
DATA -2,-5,-6,-1.2,-1.5,-6.7,4.5,8.1
DATA 5,-2.3,4,5.6,12.2,18.2,14.1,16
DATA 4.1,5.2,16.7,18.4,19.1,20.2,13.6,14.8
DATA 15.2,-1.8,13.6,-4.9,5.4,19.8,16.4,-20.9
DATA -3.6,6,-8.2,-9.1,4,-2.5,2,3.4
DATA 4.7,8.3,9.4,10.5,11,19,15.4,18.9
```

```
//
MAT READ a()
MAT CPY b()=a()              // save the original
//                              matrix
PRINT "Original matrix"
PRINT
MAT PRINT a(),7,2
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
//
//  row-wise normalising
//
CLS
MAT NORM a(),0
PRINT
PRINT "Row-wise normalised: "
PRINT
MAT PRINT a(),7,2
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
//
//  testing of the row-wise normalising
//
PRINT
PRINT "Test: "
PRINT
FOR i%=1 TO n%
  MAT XCPY v()=a(i%,1)       // copies a() row-wise
  //                            into vector v()
  MAT MUL x=v()*v()          // calculates the scalar
  //                            product of v() and v()
  PRINT x'
NEXT i%
PRINT
PRINT
```

```
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
//
//  column-wise normalising
//
CLS
MAT CPY a()=b()              // copy the original
//                              matrix back
MAT NORM a(),1
PRINT "Column-wise normalised: "
PRINT
MAT PRINT a(),7,2
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
//
//  testing of column-wise normalising
//
PRINT
PRINT "Test: "
PRINT
FOR i%=1 TO n%
  MAT CPY v()=a(1,i%)        // copies a() column
  //                            wise into vector v()
  MAT MUL x=v()*v()          // calculates the
  //                            scalar product
  //                            between v() and v()
  PRINT x'
NEXT i%
```

displays

Original matrix

1.00,	2.00,	3.00,	4.00,	5.00,	6.00,	7.00,	8.00
3.20,	4.00,	-5.00,	2.40,	5.10,	6.20,	7.20,	8.10
-2.00,	-5.00,	-6.00,	-1.20,	-1.50,	-6.70,	4.50,	8.10
5.00,	-2.30,	4.00,	5.60,	12.20,	18.20,	14.10,	16.00
4.10,	5.20,	16.70,	18.40,	19.10,	20.20,	13.60,	14.80
15.20,	-1.80,	13.60,	-4.90,	5.40,	19.80,	16.40,	-20.90
-3.60,	6.00,	-8.20,	-9.10,	4.00,	-2.50,	2.00,	3.40
4.70,	8.30,	9.40,	10.50,	11.00,	19.00,	15.40,	18.90

Row-wise normalised:

0.07,	0.14,	0.21,	0.28,	0.35,	0.42,	0.49,	0.56
0.21,	0.26,	-0.32,	0.16,	0.33,	0.40,	0.47,	0.52
-0.14,	-0.35,	-0.42,	-0.08,	-0.11,	-0.47,	0.32,	0.57
0.16,	-0.07,	0.13,	0.18,	0.38,	0.57,	0.44,	0.50
0.10,	0.12,	0.39,	0.43,	0.45,	0.47,	0.32,	0.35
0.38,	-0.05,	0.34,	-0.12,	0.14,	0.50,	0.41,	-0.53
-0.23,	0.39,	-0.53,	-0.59,	0.26,	-0.16,	0.13,	0.22
0.13,	0.22,	0.25,	0.28,	0.30,	0.51,	0.42,	0.51

Test:

1 1 1 1 1 1 1 1

```
Column-wise normalised:

 0.06,  0.15,  0.11,  0.16,  0.18,  0.15,  0.22,  0.21
 0.18,  0.29, -0.19,  0.10,  0.19,  0.15,  0.23,  0.21
-0.11, -0.37, -0.23, -0.05, -0.06, -0.17,  0.14,  0.21
 0.28, -0.17,  0.15,  0.23,  0.45,  0.45,  0.44,  0.42
 0.23,  0.38,  0.63,  0.74,  0.71,  0.50,  0.43,  0.39
 0.85, -0.13,  0.51, -0.20,  0.20,  0.49,  0.51, -0.54
-0.20,  0.44, -0.31, -0.37,  0.15, -0.06,  0.06,  0.09
 0.26,  0.61,  0.35,  0.43,  0.41,  0.47,  0.48,  0.49

Test:

1 1 1 1 1 1 1 1
```

Each square matrix with real or complex elements can be integrated into a real or a complex number, which is called the determinant of a matrix. The determinant is a function of the elements of a matrix.

Let's assume that a() is a square matrix with n rows and n columns. To calculate its determinant, all possible products are created from elements in a() in such a way, that each of the products from every row and column contains exactly one element as a factor. The normal order of the row indices can be maintained by appropriately selecting the factors in all of the products:

```
a(1, )*a(2,. )*...*a(n, )
```

Following that, the column indices are used to create all possible n! permutations and the numbers in each permutation are then used as column indices.

The column indices in a square matrix of (3,3) type can in this way get the values 1,2,3. This allows for 3! = 6 possible permutations of the column indices:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Since the row indices are left in their normal order, the following six products result:

```
a(1,1)*a(2,2)*a(3,3)
a(1,1)*a(2,3)*a(3,2)
a(1,2)*a(2,1)*a(3,3)
a(1,2)*a(2,3)*a(3,1)
a(1,3)*a(2,1)*a(3,2)
a(1,3)*a(2,2)*a(3,1)
```

The number of available inversions k must further be calculated for each permutation of the column indices. An inversion is the count of elements out of order in a permutation. The number of the inversions k for the above permutations is as follows:

Permutation	**Number of inversions k**
1 2 3	k = 0
1 3 2	k = 1 (since 2 follows 3)
2 1 3	k = 1 (since 1 follows 2)
2 3 1	k = 2 (since 1 follows 2 and 3)
3 1 2	k = 2 (since 1 follows 3 and 2 follows 3)
3 2 1	k = 3 (since 2 follows 3 and 1 follows 3 and 2)

Each of the n! products is then (-1)^k multiplied, where k is the number of inversions for the corresponding column permutation. Finally all of this products are added up.

The determinant of a square matrix of the (3,3) type is therefore gives as:

```
DET = (-1)^0*a(1,1)*a(2,2)*a(3,3) + (-1)^1*a(1,1)*a(2,3)*a(3,2) +
      (-1)^1*a(1,2)*a(2,1)*a(3,3) + (-1)^2*a(1,2)*a(2,3)*a(3,1) +
      (-1)^2*a(1,3)*a(2,1)*a(3,2) + (-1)^3*a(1,3)*a(2,2)*a(3,1).
```

If the determinant of a matrix is not equal to 0, the matrix is regular, otherwise it is singular.

The GFA-BASIC command **MAT DET x=a([i,j])[,n]** calculates the determinant of a real number matrix a() or the determinant of a square section of this matrix. When the section is selected, it starts at i,j row and column offset, and has the height and width of n elements.

Example:

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8),b(4,4)
MAT READ a()
PRINT "Original matrix: "
PRINT
MAT PRINT a(),5,2        // original matrix
PRINT
MAT DET x=a()            // calculates the
//                          determinant
PRINT "Determinant = ";x
```

```
PRINT
MAT DET y=a(3,2),4      // calculates the determinant
//                         of a matrix segment
PRINT "Segment determinant = ";y
PRINT
PRINT "Test:"
PRINT
PRINT "Matrix segment:"
PRINT
MAT CPY b()=a(3,2),4,4
MAT PRINT b(),5,2
PRINT
MAT DET z=b()
PRINT "Determinant = ";z
```

displays

```
Original matrix:

 2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
 3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
 1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
 4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
 2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
 2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00

Determinant         = -15045648.8341
Segment determinant =  1337.7709
```

```
Matrix segment:

 2.30, 6.00, 3.20, 6.00
 5.00, 6.00, 8.20, 4.10
 2.30, 9.00, 8.10, 0.00
 7.10, 8.30, 9.10,-5.00

Determinant = 1337.7709
```

The GFA-BASIC command **MAT QDET** can also be used to calculate the determinant. It differs from **MAT DET** in that **MAT DET** is optimised for accuracy while **MAT QDET** is optimised for speed. If the determinant approaches 0, **MAT DET** should be used. Otherwise both **MAT DET** and **MAT QDET** return the same result.

Example:

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8)
MAT READ a()
PRINT "Original matrix:"
PRINT
MAT PRINT a(),5,2          // original matrix
PRINT
MAT DET x=a()              // calculates the
//                            determinant
PRINT "Determinant with MAT DET = ";x
PRINT
MAT QDET y=a()             // calculate the deter-
//                            minant
```

```
PRINT "Determinant with MAT QDET = ";y
PRINT
PRINT "Deviation = ";x-y
```

displays

```
Original matrix:

 2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
 3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
 1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
 4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
 2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
 2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00

Determinant with MAT DET  = -15045648.8341
Determinant with MAT QDET = -15045648.8341

Deviation = 0
```

A rank of any matrix is a positive whole number. If a() is of type (0,0), its rank is a() = 0. If a() is of type (m,n), its rank equals the maximum number of linearly independent rows (vectors).

If an n-order square matrix has a rank of n, its determinant is not zero, i.e. the matrix is regular. Otherwise its determinant equals zero, i.e. the matrix is singular.

The GFA-BASIC command **MAT RANK** calculates the rank of an n-order square matrix. As with **MAT DET** and **MAT QDET**, any row or column offset can be used. The number of elements per line of a matrix segment is

defined with n. Internally, this creates a segment matrix from i-th row and j-th column.

Example:

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8),b(4,4)
MAT READ a()
PRINT "Original matrix:"
PRINT
MAT PRINT a(),5,2       // original matrix
PRINT
MAT RANK x=a()          // calculates the rank
PRINT
PRINT "Rank = ";x
MAT CPY b()=a(3,2),4,4
PRINT "Segment matrix :"
PRINT b(),5,2
MAT RANK y=a(3,2),4
PRINT
PRINT "Rank of the segment matrix = ";y
```

```
displays

Original matrix:

 2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
 3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
 1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
 4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
 2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
 2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00

Rank = 8

Segment matrix:

 2.30, 6.00, 3.20, 6.00
 5.00, 6.00, 8.20, 4.10
 2.30, 9.00, 8.10, 0.00
 7.10, 8.30, 9.10,-5.00

Rank of the segment matrix = 4
```

An example of a singular matrix is when, for example, the second row of the original matrix is replaced by the triplicate of the first row. In such a case, the rank of the matrix in the above example equals 7, since the first and the second row are linearly dependent (second row = 3 * first row). The determinant of this matrix is then equal to 0.

Example:

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA 6,13.5,18,9.6,21,5.1,-12,36
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8),b(4,4),c(4,4)
MAT READ a()
PRINT "Original matrix:"
PRINT
MAT PRINT a(),5,2
MAT RANK x=a()
PRINT
PRINT "Rank = ";x
MAT DET y=a()
PRINT
PRINT "Determinant = ";y
```

displays

Original matrix:

```
 2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
 6.00,13.50,18.00, 9.60,21.00, 5.10,12.00,36.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
 3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
 1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
 4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
 2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
 2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00

Rank        = 7
Determinant = 0
```

You must have noticed by now that no mention of matrix division has been made so far. This is because there is no definition of matrix division. Instead, the linear algebra has the multiplication of a matrix with a so-called inverse of another matrix.

The inversion is only possible for n-order regular (determinant not equal 0) square matrices. If the matrix is singular (determinant = 0), it can't be inverted. An inversion of an n-order square matrix a() is an n-order square matrix b(), for which the matrix products

```
a()*b() and
b()*a()
```

produce an identical n-order uniform matrix, i.e.

```
a()*b()=b()*a() =  1,0,0,...,0
                   0,1,0,...,0
                   .
                   0,0,0,...,1
```

The GFA-BASIC command **MAT INV b()=a()** calculates the inverse of an n-order regular matrix.

Example:

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8),b(8,8),c(8,8),d(8,8)
MAT READ a()
```

```
PRINT "Original matrix a() :
PRINT
MAT PRINT a(),5,2
PRINT
MAT INV b()=a()         // calculate the inverse
PRINT
PRINT "Inverse of a()" :
PRINT
MAT PRINT b(),6,3
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
CLS
PRINT "b()*a() :"
PRINT
MAT MUL c()=b()*a()
MAT PRINT c(),6,3
PRINT
PRINT "a()*b() :"
PRINT
MAT MUL d()=a()*b()
MAT PRINT d(),6,3
```

displays

Original matrix a():

```
 2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
 3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
 1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
 4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
 2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
 2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00
```

Inverse of a():

```
-0.337,-0.127, 0.058, 0.182, 0.041,-0.215, 0.276, 0.191
-0.400,-0.183,-0.083, 0.252, 0.037,-0.297, 0.432, 0.375
 0.648, 0.378, 0.100,-0.401,-0.201, 0.605,-0.647,-0.547
-0.098,-0.129,-0.033, 0.052, 0.183,-0.146, 0.080, 0.142
 0.331, 0.166, 0.068,-0.199,-0.041, 0.195,-0.366,-0.186
-0.387,-0.208,-0.044, 0.167, 0.198,-0.371, 0.401, 0.336
-0.259,-0.072,-0.020, 0.186, 0.043,-0.184, 0.209, 0.136
-0.233,-0.156,-0.060, 0.216, 0.041,-0.239, 0.321, 0.169
```

b()*a():

```
 1.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000,-0.000
 0.000, 1.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000
-0.000,-0.000, 1.000,-0.000,-0.000,-0.000,-0.000,-0.000
 0.000, 0.000, 0.000, 1.000, 0.000,-0.000, 0.000,-0.000
-0.000,-0.000,-0.000, 0.000, 1.000,-0.000,-0.000,-0.000
 0.000, 0.000, 0.000, 0.000, 0.000, 1.000, 0.000, 0.000
 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 1.000, 0.000
 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 1.000
```

a()*b():

```
 1.000,-0.000,-0.000, 0.000, 0.000,-0.000, 0.000, 0.000
 0.000, 1.000, 0.000,-0.000, 0.000, 0.000,-0.000, 0.000
 0.000, 0.000, 1.000,-0.000, 0.000, 0.000,-0.000, 0.000
-0.000,-0.000,-0.000, 1.000, 0.000,-0.000, 0.000, 0.000
 0.000, 0.000, 0.000,-0.000, 1.000, 0.000,-0.000, 0.000
-0.000, 0.000, 0.000, 0.000,-0.000, 1.000,-0.000, 0.000
-0.000,-0.000, 0.000,-0.000,-0.000,-0.000, 1.000, 0.000
 0.000, 0.000, 0.000,-0.000,-0.000, 0.000,-0.000, 1.000
```

A frequent application of matrix calculations is for solving of multiple linear equations. Such equations can be described as a system of m linear equations with n variables x(1), x(2), ... , x(n):

```
a(1,1)*x(1)+a(1,2)*x(2)+ ... +a(1,n)*x(n) = b(1)
a(2,1)*x(1)+a(2,2)*x(2)+ ... +a(2,n)*x(n) = b(2)
.          .             ... .             = .
a(m,1)*x(1)+a(m,2)*x(2)+ ... +a(m,n)*x(n) = b(m).
```

a(i,j); i = 1,...,m; j = 1,...,n are called coefficients, b(i) are called the absolute components and x(j) are called the solution components of a linear equation system.

If a() is a matrix of (m,n) type, x() an n-dimensioned and b() an m-dimensioned column vector, the equation system can also be written as:

```
a()*x()*b()
```

where a() is the coefficient matrix, b() the result and x() the solution vector of the system.

A special case occurs when m = n since the equation system can be solved by multiplying both sides of the equation with the inverse of the coefficient matrix. If u() is the inverse of a() and i() an n-order uniform matrix, it follows that

```
u()*a()*x() = u()*b(); u()*a() = i()
i()*x()     = u()*b(); i()*x() = x()
x()         = u()*b()
```

The solution vector is obtained by multiplying the result vector with the inverse of the coefficient matrix.

Example:

A group of 3076 men between the ages of 16 and 65 are analysed in an insurance company. The individuals are divided in four age groups. Four risk factors are then created for each age group to asses

1. the average number of accidents reported to the doctor over the last five years,

2. the average of reported sick leave days over the last two years,

3. the average number of days on which a medication was taken during the last year and

4. the average number of cigarettes smoked daily.

This results in the following:

	Accidents	Days sick	Medication	Cigarettes
Group 1:	17.3	7.4	52.5	17.6
Group 2:	22.6	19.1	67.3	25.2
Group 3:	67.2	12.2	43.1	23.0
Group 4:	12.4	45.2	73.2	20.3

These values are entered in the coefficient matrix a().

Furthermore, during the last year each of the 4 groups accounted, on the average, for the following medical expenses:

Group 1:	3500
Group 2:	2700
Group 3:	4300
Group 4:	6100

This data comprise the solution vector b().

Assuming that the medical expenses can be projected as a linear combination of four risk classes (accidents, sick leave, medications and cigarettes) the following equation system can be set up

```
a()*x()=b()
```

to determine the solution vector x().

Example program:

```
a:
DATA 17.3,7.4,52.5,17.6
DATA 22.6,19.1,67.3,25.2
DATA 67.2,12.2,43.1,23.0
DATA 12.4,45.2,73.2,20.3
b:
DATA 3500,2700,4300,6100
DIM a(4,4),b(4,1),x(4,1),u(4,4)
RESTORE a
MAT READ a()
RESTORE b
MAT READ b()
PRINT "Coefficient matrix a():"
MAT PRINT a(),5,1
PRINT
PRINT "Result vector b():"
MAT PRINT b(),5,0
MAT INV u()=a()
PRINT
PRINT "Inverse of a()=u():"
MAT PRINT u(),6,3
MAT MUL x()=u()*b()
PRINT
PRINT "Solution vector x():"
MAT PRINT x(),8,3
```

displays

```
Coefficient matrix a():

17.3,  7.4, 52.5, 17.6
22.6, 19.1, 67.3, 25.2
67.2, 12.2, 43.1, 23.0
12.4, 45.2, 73.2, 20.3

Result vector b():

3500
2700
4300
6100

Inverse of a()=u():

 0.035,-0.053, 0.022, 0.011
-0.061, 0.020, 0.005, 0.022
 0.106,-0.091,-0.001, 0.021
-0.269, 0.313,-0.021,-0.083

Solution vector x():

 138.641
  -2.384
 252.867
-690.702
```

In many areas of mathematics, especially in applied mathematics, the matrix calculations include the calculation of individual vectors and values. Due to space constraints, these routines were not implemented in GFA-BASIC. For all of those who painfully miss these routines here's a program to calculate individual vectors and values for real symmetrical matrices (the Jacobi method).

a() is an n-order symmetrical matrix with real elements, x() is an n-dimensioned vector and lambda is a scalar.

Each vector x(), for the equation

```
a()*x() = lambda*x()
```

is the vector to calculate in matrix a() and lambda is the a() value that belongs to x(). Every n-order matrix contains exactly n values (which are not necessarily unique).

Two rules apply for these values in a matrix:

1. The sum of values in a() is equal to the track of a().
2. The product of n values of a() equals the determinant of a().

During the calculation both of these rules can be used to check the accuracy.

Demo program:

```
DATA 1,0.235,0.416,0.523,0.765
DATA 0.235,1,0.543,0.881,0.721
DATA 0.416,0.543,1,0.419,0.213
DATA 0.523,0.881,0.419,1,0.642
DATA 0.765,0.721,0.213,0.642,1
//
DIM b(5,5),a(5,5),r(5),u(5,5),x(5,5),y(5,5)
//
MAT READ b()
PRINT "Original matrix:"
PRINT
MAT PRINT b(),5,2
mo__%=1              // = 1 for MAT BASE 1, otherwise 0
@mat_pca(20,0.003,r(),a(),b())
//
```

```
PRINT
PRINT "Value :"
PRINT
MAT PRINT r(),7,4
PRINT
PRINT "Vectors :"
PRINT
MAT PRINT a(),7,4
//
MAT DET x=b()
u=0
v=0
y=1
FOR i%=1 TO 5
  ADD v,r(i%)
  ADD u,b(i%,i%)
  MUL y,r(i%)
NEXT i%
PRINT
PRINT "Determinant of b()   : ";x
PRINT "Product of values : ";y
PRINT "Track of b()           : ";u
PRINT "Sum of values    : ";v
//
// --------------------------------------------------
// All real values of a symmetric matrix b() with
// normalised vectors (Jacobi method)
//
// Global variables: no_go__!  (are set in PROCEDURE chk_mat.)
//
// call: @mat_pca(iter,eps,r(),a(),b())
//
// r()  = vector b()
// a()  = Matrix of normalised vectors belonging to r()
//        (column-wise)
// b()  = original matrix
// iter = number of iterations
```

```
// eps  = cancel criterion
//
PROCEDURE mat_pca(it%,eps,VAR r(),a(),b())
  //
  LOCAL i_%,j_%,k_%,l_%,n_%
  LOCAL az_%,as_%,adim_%,bz_%,bs_%,bdim_%,lz_%,ls_%,ldim_%
  LOCAL s_,s1_,s2_,s3_,s4_,s5_,s6_,s7_,xmin_
  //
  @chk_mat(a(),az_%,as_%,adim_%)              // testing of matrix a()
  IF NOT no_go__!                             // a() is dimensioned
    IF adim_%=2                               // a() is two-dimensional
      @chk_mat(b(),bz_%,bs_%,bdim_%)          // testing of matrix b()
      IF NOT no_go__!                         // b() is dimensioned
        IF bdim_%=2                           // b() is two-dimensional
          IF (bz_%=bs_%) AND (az_%=as_%)      // a() and b() must
            //                                   be square
            IF az_%=bz_%                      // a() and b() must be of
              //                                 the same order
              @chk_mat_sym(b())               // testing of b() for
              //                                 symmetry
              IF NOT as__!                    // symmetrical
                //
                @chk_mat(r(),lz_%,ls_%,ldim_%)// testing of
                //                               matrix r()
                IF NOT no_go__!               // r() is dimensioned
                  IF ldim_%=1                 // r() is dimensioned as a
                    //                           vector
                    IF lz_%=az_%
                      it%=MAX(it%,10)
                      eps=ABS(eps)
                      IF eps=0
                        eps=0.0001
                      ENDIF
                      //
                      FOR i_%=mo__% TO az_%
                        r(i_%)=b(i_%,i_%)     // assign
                      NEXT i_%                // value vectors
```

```
//
MAT ONE a()
//
DIM d_(az_%),inf_(az_%) // interim vectors
DIM im_(az_%,az_%)      // working copy of
//                        matrix b()
//
MAT CPY im_()=b()       // save matrix b()
//
FOR k_%=1 TO it%        // iteration
  s_=0
  FOR i_%=mo__% TO az_%-1
    FOR j_%=i_%+1 TO az_%
      ADD s_,ABS(im_(i_%,j_%))
    NEXT j_%
  NEXT i_%
  //
  EXIT IF s_=0
  //
  IF k_%<4
    xmin_=s_/(5*az_%*az_%)
  ELSE
    xmin_=0
  ENDIF
  //
  MAT CPY inf_()=r()
  MAT CLR d_()
  //
  FOR i_%=mo__% TO az_%-1  // main loop
    //
    FOR j_%=i_%+1 TO az_%
      //
      s1_=100*ABS(im_(i_%,j_%))
      //
      IF k_%>4
        IF ABS(r(i_%))+s1_=ABS(r(i_%))
          IF ABS(r(j_%))+s1_=ABS(r(j_%))
```

```
        im_(i_%,j_%)=0
      ENDIF
    ENDIF
  ENDIF
  //
  IF ABS(im_(i_%,j_%))>xmin_
    //
    s2_=r(j_%)-r(i_%)
    //
    IF ABS(s2_)+s1_=ABS(s2_)
      s3_=im_(i_%,j_%)/s2_
    ELSE
      //
      s4_=s2_/(2*im_(i_%,j_%))
      s3_=1/(ABS(s4_)+SQR(1+s4_*s4_))
      //
      IF s4_<0
        MUL s3_,-1
      ENDIF
    //
    ENDIF
    //
    s5_=1/SQR(1+s3_*s3_)
    s6_=s3_*s5_
    s7_=s6_/(s5_+1)
    s2_=s3_*im_(i_%,j_%)
    //
    SUB d_(i_%),s2_
    ADD d_(j_%),s2_
    SUB r(i_%),s2_
    ADD r(j_%),s2_
    im_(i_%,j_%)=0
    //
    FOR l_%=mo__% TO i_%-1
      s1_=im_(l_%,i_%)
      s2_=im_(l_%,j_%)
      im_(l_%,i_%)=s1_-s6_*(s2_+s1_*s7_)
```

```
          im_(l_%,j_%)=s2_+s6_*(s1_-s2_*s7_)
        NEXT l_%
        //
        FOR l_%=i_%+1 TO j_%-1
          s1_=im_(i_%,l_%)
          s2_=im_(l_%,j_%)
          im_(i_%,l_%)=s1_-s6_*(s2_+s1_*s7_)
          im_(l_%,j_%)=s2_+s6_*(s1_-s2_*s7_)
        NEXT l_%
        //
        FOR l_%=j_%+1 TO az_%
          s1_=im_(i_%,l_%)
          s2_=im_(j_%,l_%)
          im_(i_%,l_%)=s1_-s6_*(s2_+s1_*s7_)
          im_(j_%,l_%)=s2_+s6_*(s1_-s2_*s7_)
        NEXT l_%
        //
        FOR l_%=mo__% TO az_%
          s1_=a(l_%,i_%)
          s2_=a(l_%,j_%)
          a(l_%,i_%)=s1_-s6_*(s2_+s1_*s7_)
          a(l_%,j_%)=s2_+s6_*(s1_-s2_*s7_)
        NEXT l_%
        //
      ENDIF
      //
    NEXT j_%
    //
  NEXT i_%
  //
  FOR i_%=mo__% TO az_%
    r(i_%)=inf_(i_%)+d_(i_%)
  NEXT i_%
  //
NEXT k_%
//
ERASE d_(),inf_(),im_()
```

```
                //
              ELSE
                @error_code(1)          // values or vectors could
                //                         not be determined
              ENDIF
              //
            ELSE
              @error_code(2)            // a vector, not a matrix
              //
            ENDIF
            //
          ENDIF                         // r() is dimensioned
          //
        ELSE
          @error_code(3)                // the matrix is not
          //                               symmetrical
        ENDIF
        //
      ELSE
        @error_code(1)                  // values or vectors could
        //                                 not be determined
      ENDIF
      //
    ELSE
      @error_code(4)                    // not a square matrix
    ENDIF
    //
  ELSE
    @error_code(5)                      // not a two-dimensional
    //                                     matrix
  ENDIF
  //
ENDIF                                   // b() is dimensioned
//
ELSE
  @error_code(5)                        // not a two-dimensional
  //                                       matrix
```

```
    ENDIF
    //
  ENDIF                                        // a() is dimensioned
  //
  //
RETURN
// ----------------------------------------------------------------
// testing of matrix a()
//
// z% returns the number of rows,
// s% returns the number of columns and dim% the
// dimensions of the matrix.
//
PROCEDURE chk_mat(VAR a(),z%,s%,dim%)
  //
  LOCAL v_%
  ERASE check_option_base&()                   // determine OPTION BASE 0
  //                                              or 1
  DIM check_option_base&(1)
  //
  no_go__!=FALSE
  //
  v_%={{*check_option_base&()}}-1              // v_%=1 for OPTION BASE
  //                                              0; otherwise 0
  ERASE check_option_base&()
  //
  dim%=INT{*a()+4}                             // dim% = dimensions of
  //                                              the matrix
  //
  SELECT dim%                                  // only one- and two-
  //                                              dimensional matrices
  //                                              are allowed
  CASE 1                                       // one-dimensional
    //
    z%={{*a()}}-v_%
    s%=1
    //
```

```
  CASE 2                                    // two-dimensional
    //
    z%={{*a()}+4}-v_%
    s%={{*a()}}-v_%
    //
  DEFAULT                                   // error message if matrix
    //                                         is not one- or two-
    //                                         dimensional
    @error_code(1)
    no_go__!=TRUE
    //
  ENDSELECT
  //
RETURN
// ---------------------------------------------------------------
// testing of matrix a() for symmetry
//
// Global variable: as__! (will be set if a() is not symmetrical)
//
PROCEDURE chk_mat_sym(VAR a())
  //
  LOCAL i_%,j_%,az_%,as_%,adim_%
  //
  @chk_mat(a(),az_%,as_%,adim_%)
  '
  as__!=FALSE
  //
  IF adim_%=2
    //
    IF az_%=as_%
      //
      FOR i_%=mo__% TO az_%-1
        FOR j_%=i_%+1 TO az_%
          EXIT IF a(j_%,i_%)<>a(i_%,j_%)
        NEXT j_%
        //
        EXIT IF j_%<az_%+1
```

```
      NEXT i_%
      //
      IF (j_%<az_%+1) OR (i_%<az_%)
        as__!=TRUE
      ENDIF
      //
    ELSE
      as__!=TRUE
    ENDIF
    //
  ELSE
    as__!=TRUE
    @error_code(5)
  ENDIF
RETURN
// -----------------------------------------------------------------------
// table of error messages
//
PROCEDURE error_code(ec__%)
  //
  LOCAL r_$,a_$,d_%
  //
  r_$="Return"
  SELECT ec__%
  CASE 1
    a_$="Values or vectors|can't be determined"
  CASE 2
    a_$="A vector, not a matrix"
  CASE 3
    a_$="Matrix not symmetrical"
  CASE 4
    a_$="Matrix not square"
```

```
  CASE 5
    a_$="Matrix not two-dimensional"
  CASE 6
    a_$="Matrix operations for |one- or dimensional|arrays only"
  ENDSELECT
  ALERT 3,a_$,1,r_$,d_%
RETURN
// ------------------------------------------------------------------
```

displays

Original matrix:

```
 1.00, 0.24, 0.42, 0.52, 0.77
 0.24, 1.00, 0.54, 0.88, 0.72
 0.42, 0.54, 1.00, 0.42, 0.21
 0.52, 0.88, 0.42, 1.00, 0.64
 0.77, 0.72, 0.21, 0.64, 1.00
```

Values:

```
 0.2498, 3.1888, 0.8991,-0.0952, 0.7576
```

Vectors:

```
 0.2323, 0.4050,-0.6115,-0.4617, 0.4414
-0.2792, 0.4879, 0.4207,-0.6142,-0.3603
-0.1818, 0.3406, 0.4923, 0.2587, 0.7360
 0.7337, 0.4997, 0.1598, 0.3336,-0.2742
-0.5447, 0.4820,-0.4256, 0.4810,-0.2420
```

```
Determinant of b(): -0.05166800663097
Product of values:  -0.05166800663096
Track of b():        5
Sum of values:       5
```

2.6 Graphics

A whole range of graphic commands and functions are implemented in GFA-BASIC. They can be divided into

1. commands and functions for character graphics

2. commands and functions for pixel graphics

3. commands and functions for the creation of a (pixel) graphic user interface

4. commands and functions for the management of the (pixel) graphic user interface.

The groups 1 and 2 are addressed in this paragraph. For groups 3 and 4 please refer to the chapter "The commands and functions for the creation and management of a (pixel) graphic user interface".

All four command categories assume that you have a graphic card (hardware) which supports the relevant functions. To better understand the various graphic cards we have present here the following summary:

2.6.1 Text graphic cards

This type of a graphic card is controlled by the operating system with the BIOS (Basic Input Output System). There are two video cards of this type:

- monochrome (IBM) video card (MDA) and
- monochrome (Hercules) graphic card (HGA)

2.6.1.1 The monochrome (IBM) video card (MDA)

This card has the resolution of 25 lines and 80 columns. The coordinates on this card (line 0, column 0) originate in the upper left corner of the screen. The lines run top to bottom from 0 to 24. The columns run left to right from 0 to 79. This card type is mainly used with monchrome monitors. The starting address for this card in MS-DOS RAM is at $B000:0000.

The graphics on this card are limited to the graphic characters from the IBM character set (refer to the relevant table in the Reference Guide appendix). This card is invoked in GFA-BASIC with the **SCREEN 7** command.

2.6.1.2 The monochrome (Hercules) graphic card (HGA)

Although this card contains two screen pages with the resolution of 25 lines by 80 columns and therefore two graphic pages with the resolution of 730x348 pixels, it is nevertheless controlled by the BIOS as the card described under A1.

The text mode of this card is invoked in GFA-BASIC with the **SCREEN 7** command, and the graphic mode with the GFA-BASIC command **SCREEN 1.**

2.6.1.3 The (IBM) colour graphic card (CGA)

This card offers all features described under A1 as well as the option of displaying text in 40x25 raster. The individual characters are then shown twice as wide as in the 80x25 raster. In the 40x25 resolution the BIOS can use eight pages while in the 80x25 resolution it can use four screen pages.

In the text mode the GFA-BASIC **TCOLOR** command can be used to set both the background and the foreground colours to one of eight available colours (see **TCOLOR** in the Reference Guide).

In addition, this card offers various options for pixel graphics. There is a graphic mode with the resolution of 640x200 pixels and a resolution of 320x200 points.

By using the GFA-BASIC **COLOR** command in the 320x200 resolution one of 16 colours can be used as the background colour. Two palettes are available for the foreground colour, one with turquoise (cyan), purple (magenta) and white, and the other with green, red and yellow. Only four colours can be used simultaneously on the screen.

In the 640x200 resolution, the GFA-BASIC **COLOR** command can use one of 16 colours as either background or foreground colours. In contrast to the 320x200 pixel resolution only two colours can be simultaneously.

The starting address for this card type is in MS-DOS RAM at $B800:0000.

The GFA-BASIC calls for the text mode on this card are:

SCREEN 3	for the 80x25 and
SCREEN 1	for the 40x25 resolution.

The graphic resolutions are invoked with the GFA-BASIC commands

SCREEN 6	(640x200) and
SCREEN 5	(320x200)

The special GFA-BASIC text/graphic commands implemented for the cards discussed in A1, A2 and A3 are also valid for the card types described below.

2.6.2 The commands and functions for character graphics

TCOLOR
TBOX
TPBOX
TCLIP
TCLIP OFF
SCROLL ON
SCROLL OFF
WRAP ON
WRAP OFF
TGET
TPUT

TCOLOR sets the character attributes for text output in text mode (PRINT). **TCOLOR** must be followed by a parameter which defines the attribute byte for the card in question.

For **MDA resolution** (A1) this byte is assigned as follows:

Bit	7	6	5	4	Background
	1	x	x	x	blinking
	0	x	x	x	not blinking
	x	0	0	0	black background
	x	1	1	1	white background

Bit	3	2	1	0	Foreground
	1	x	x	x	high intensity
	0	x	x	x	normal intensity
	x	0	0	0	black characters
	x	0	0	1	underlined
	x	1	1	1	white characters

For standard **CGA resolution** (A3):

Bit	7	6	5	4	Background
	1	x	x	x	blinking
	0	x	x	x	not blinking
x	0	0	0	black	
	x	0	0	1	blue
	x	0	1	0	green
	x	0	1	1	turquoise (cyan)
	x	1	0	0	red
	x	1	0	1	purple (magenta)
	x	1	1	0	brown
	x	1	1	1	white

Bit	3	2	1	0	Foreground
	1	x	x	x	high intensity
	0	x	x	x	normal intensity
	0	0	0	0	black
	0	0	0	1	blue
	0	0	1	0	green
	0	0	1	1	turquoise (cyan)
	0	1	0	0	red
	0	1	0	1	purple (magenta)
	0	1	1	0	brown
	0	1	1	1	white
	1	0	0	0	grey
	1	0	0	1	light blue
	1	0	1	0	light green
	1	0	1	1	light turquoise
	1	1	0	0	pink
	1	1	0	1	purple pink
	1	1	1	0	yellow
	1	1	1	1	bright white

It's much clearer when these parameters are specified in binary. For example **TCOLOR %10000111** sets on an MDA card the blinking character in white on black background.

TCOLOR %00001011 sets in a CGA card the characters in light turquoise on black background.

Example:

```
SCREEN 7                    //      MDA resolution
TCOLOR %10000111
PRINT "Hi there! This is Eddie, your shipboard
       computer,"
PRINT "and I'm feeling just great, guys, and I know
       I'm"
PRINT "just going to get a bundle of kicks out of
       any"
PRINT "program you care to run trough me"
```

Prints a passage from chapter 16 in D. Adams, "The Hitchhiker's Guide to the Galaxy", Pocket Books, N.Y., 1981.

Example:

```
SCREEN 3                  // 80x25 resolution
TCOLOR %00001011
PRINT "Hi there! This is Eddie, your shipboard
       computer,"
PRINT "and I'm feeling just great, guys, and I know
       I'm"
PRINT "just going to get a bundle of kicks out of
       any"
PRINT "program you care to run trough me."
```

Displays the same passage in 80x25 resolution in turquoise on a black background.

TBOX draws a frame on the screen with the characters from the IBM character set. The command requires five parameters: n,sz,ss,ez and es.

The parameter n determines the frame type as follows:

n = 0	frame out of space characters
n = 1	single line frame
n = 2	double line frame

The parameters sz and ss determine the start of the frame, whereby sz specifies the starting line and ss the starting column.

The parameters ez and es determine the end of the frame, whereby ez specifies the end line and es the end column.

If either ez-sz or es-ss are negative, a frame with the maximum number of lines/columns is drawn.

Example:

```
SCREEN 7                    // MDA resolution
TBOX 1,0,0,79,10
```

Draws a frame out of single white lines on black background in the upper half of the screen.

Since this frame is drawn with characters from the IBM graphic character set, the frame colour can be selected - depending on the card used - with the **TCOLOR** command.

Example:

```
SCREEN 3                    // 80x25 characters
TCOLOR %00000011
TBOX 1,0,0,79,10
```

Draws a frame out of single turquoise line characters on a black background in the upper half of the screen.

TPBOX uses the same parameters as **TBOX** and, without the **TCOLOR** command, it also works line **TBOX.** But when **TCOLOR** is used with **TPBOX** the area inside the frame is painted with the colour specified in **TCOLOR.**

Example:

```
SCREEN 3                    // 80x25 character
TCOLOR %00011110
TPBOX 2,0,0,79,10
```

Draws a yellow box on blue background in the upper half of the screen.

TCLIP limits all text output to a clipping rectangle. The GFA-BASIC command **TCLIP** has two forms:

TCLIP sz,ss TO ez,es and
TCLIP sz,ss,as,az

Both variations require four parameters, whereby the first two, sz and ss, are the same for both. They specify the starting line and column for the clipping area. The **TCLIP sz,ss TO ez,es** variant specifies in ez the end line and in es the end column.

For **TCLIP sz,ss,as,az** the number of lines is specified in as and the number of columns in az. The clipping rectangle extends starting from sz and ss. The invalid values cause **TCLIP** to act the same as **TBOX** or **TPBOX.**

Example:

```
SCREEN 3                   // 80x25 characters
TCOLOR %00000011
TCLIP 0,0 TO 50,20
TBOX 2,0,0,50,20
PRINT "First, it is slightly cheaper;"
PRINT "and second it has the words"
PRINT "DON'T PANIC inscribed in large"
PRINT "friendly letters on its cover"
REPEAT
UNTIL LEN(INKEY$)
```

Displays a quotation from the foreword of D. ADAMS, "The Hitchhiker's Guide to the Galaxy", Pocket Books, N.Y, 1981, in a turquoise box on a black background.

The text is "clipped" within the box.

TCLIP OFF is the GFA-BASIC command to switch the text clipping off.

When the text extends beyond the right edge of the screen, the GFA-BASIC **WRAP ON** command causes the rest of the string to be automatically printed on the next line. When a GFA-BASIC program is run the **WRAP ON** is on as default.

The GFA-BASIC **WRAP OFF** command turns off the line splitting caused by **WRAP ON**. This means that the output of a line which extends beyond the right edge of the screen is stopped in the relevant column (79 or 39).

Similarly to **WRAP ON**, the **SCROLL ON** moves the screen "up" when the string output extends beyond the line 24. The whole screen is then "moved up" and the rest of the string is written on line 24 which is now free. When a GFA-BASIC program is run the **SCROLL ON** is on as default.

SCROLL OFF turns off the page scrolling set with **SCROLL ON**. Once line 24 has been used, this causes all strings to always overwrite the bottom line without previously clearing the rest of this line.

When **TCLIP** is used to turn the text clipping on, **WRAP ON**, **WRAP OFF**, **SCROLL ON** and **SCROLL OFF** are all limited to this clipping rectangle.

TGET reads a text graphic area into a string variable. The command requires five parameters: **sz,ss,ez,es,a$**.

sz and ss define the upper left corner, ez and es the lower right corner of the relevant graphic area.

a$ defines the string variable into which this are should be read in.

TPUT writes the graphic area read in with **TGET** back to the screen. **TPUT** requires three parameters: **sz,ss** and **a$**.

sz and ss specify the position for the upper left corner of the screen area.

a$ defines the string variable where the area was saved.

Example:

```
SCREEN 3                    // 80x25 characters
TCOLOR %00000100
TCLIP 0,0 TO 35,10
TBOX 2,0,0,35,10
TCOLOR %00001110
PRINT "First, it is slightly cheaper;"
PRINT "and second it has the words"
PRINT "DON'T PANIC inscribed in large"
PRINT "friendly letters on its cover"
TGET 0,0,36,11,a$
REPEAT
UNTIL LEN(INKEY$)
TPUT 40,0,a$
TPUT 0,15,a$
TPUT 40,15,a$
REPEAT
UNTIL LEN(INKEY$)
```

A red frame on the black background is drawn first in the upper left corner of the screen. The familiar text is then written in yellow on black background within this frame. The program copies the rectangle including its contents into the a$ variable and waits for a keypress. When this occurs, the rectangle is copied to the upper right, lower left and lower right corners of the screen.

2.6.3 The Pixel graphic card

The EGA and VGA graphic cards belong to this card type. In contrast to MDA, HGC and CGA video cards, they cannot be used with the ROM-BIOS routines. Instead, they all contain a special chip on the card itself which controls the traffic to and from the screen by using the co-called EGA or VGA-BIOS. The computer recognises during booting if such graphic card is available and redirects the interrupt $10 to the relevant EGA or VGA-BIOS. The problem with these graphic cards is that they are available from a number of manufacturers but there is no real standard for their usage. GFA-BASIC was tested on many EGA and VGA cards. The

implemented graphic commands were used on all tested cards without problems. However, this does not guarantee that all EGA and VGA cards are supported without problems.

2.6.3.1 The EGA card (Enhanced Graphic Adapter)

The "standard" EGA card has the resolution of 640x350 pixels and on board RAM of 256 KBytes.

From GFA-BASIC this card is invoked with **SCREEN 16.**

(Almost) all EGA cards can emulate the MDA and CGA modes without difficulties. To switch to CGA mode use **SCREEN 3** and to switch to MDA mode use **SCREEN** 7.

2.6.3.2 The VGA card (Video Graphics Array)

The "standard" VGA card has a resolution of 640x480 pixels. These days, however, there are also so-called "Super" VGA cards with the resolution of 800x600 pixels. Furthermore, the cards with a resolution of 1024x768 pixels are also beginning to appear.

All GFA-BASIC commands were tested on a video card with a resolution of 800x600. Out of the high resolution card we've had a 1024x768 card from one manufacturer made available to us.

The GFA-BASIC command to invoke the "standard" VGA card is **SCREEN 18**. For cards with higher resolutions use the **SCREEN 19** command.

For cards with resolutions higher than 800x600 GFA-BASIC uses its own port for loading of card drivers. If you have any problems with this please contact:

GFA Systemtechnik
Heerdter Sandberg 30
D-4000 Düsseldorf 11
Fax: 0211 550444

(Almost) all VGA cards can emulate the EGA, MDA and CGA modes without difficulties. To switch to EGA mode use **SCREEN 16**, to CGA mode **SCREEN 3** and to switch to MDA mode use **SCREEN 7.**

All text-graphic commands described under A can also be used in the relevant emulation mode on the EGA and VGA cards. In addition, GFA-BASIC provides a number of (pixel) graphic commands which are described below.

2.6.4 The commands and functions for pixel graphics

COLOR
GRAPHMODE
DEFFILL
_X
_Y
PLOT
PSET
DEFLINE
LINE
DRAW
SETDRAW
BOX
RBOX
PBOX
PRBOX
CIRCLE
PCIRCLE
ELLIPSE

PELLIPSE
CURVE
POLYLINE
POLYFILL
FILL
DEFTEXT
TEXT
CLIP
CLIP OFF
GET
PUT
EMSGET
EMSPUT

The **COLOR** command loads the colour registers for any subsequent graphic output. For EGA and VGA resolutions, both the fore- and background colours can be selected out of a palette with 16 colours. The GFA-BASIC command **COLOR** requires at least one parameter, which specifies the relevant colour.

This parameter always defines the foreground colour. A second parameter can optionally be specified to define the background colour. The following applies:

```
n =  0   black
n =  1   blue
n =  2   green
n =  3   turquoise (cyan)
n =  4   red
n =  5   purple (magenta)
n =  6   brown
n =  7   light grey
n =  8   dark grey
n =  9   light blue
n = 10   light green
```

```
n = 11   light turquoise
n = 12   light red
n = 13   light purple
n = 14   light yellow
n = 15   white
```

The **GRAPHMODE** defines how the output is performed in relation to the screen contents. The output is performed on bit level between the current contents of the screen and the graphic to be drawn.. **GRAPHMODE** requires one parameter n, which specifies in which operation should be performed on the graphic patterns.

Four different operations are possible:

```
1  replace
2  OR      (almost transparent)
3  XOR     (invert)
4  AND
```

The default is **GRAPHMODE 1**.

Example:

```
SCREEN 16              // EGA mode
COLOR 14
PBOX 10,10,100,200     // Graphmode 1 is default
PBOX 15,15,105,205
REPEAT                 // wait for a keypress
UNTIL LEN(INKEY$)
CLS
GRAPHMODE 2            // almost transparent
POX 10,10,100,200
COLOR 4
PBOX 15,15,105,205
REPEAT                 // wait for a keypress
UNTIL LEN(INKEY$)
CLS
```

```
GRAPHMODE 3             // invert
COLOR 14
PBOX 10,10,100,200
COLOR 4
BOX 15,15,105,205
REPEAT                  // wait for a keypress
UNTIL LEN(INKEY$)
CLS
GRAPHMODE 4             // AND operation
COLOR 14
BOX 10,10,100,200
COLOR 4
BOX 15,15,105,205
REPEAT                  // wait for a keypress
UNTIL LEN(INKEY$)
SCREEN 3
EDIT
```

Draws two boxes in yellow and, red and yellow, which overlap and the **GRAPHMODE** specified operation is performed on the overlapping area.

The **DEFFILL** defines the fill patterns for graphic commands **BOX, PCIRCLE, PELLIPSE, POLYFILL** and **FILL.**

DEFFILL requires one parameter which specifies the fill pattern.

Example:

```
SCREEN 16                // EGA mode
FOR i%=1 TO 40
  DEFFILL i%             // fill pattern
  PBOX 10,10,100,100     // rectangle
  REPEAT                 // wait for a keypress
  UNTIL LEN(INKEY$)
NEXT i%
EDIT
```

Draws repeatedly the rectangles with different fill patterns and waits for a keypress in between.

The variables **_X** and **_Y** contain the X and Y resolution of the screen when the graphic mode is on. When a window is opened **_X** and **_Y** always contain the X and Y size of the current window.

The GFA-BASIC command **PLOT** draws a graphic point in current screen colour. **PLOT** requires two parameters, whereby the first specifies the X and the second the Y coordinates in pixels of the point to drawn.

Example:

```
SCREEN 18
GRAPHMODE 3
rd%=100                  // the radius of a ball
xm%=_X/2                 // X centre of the ball
ym%=_Y/2                 // Y centre of the ball
COLOR 4                  // red
xs%=xm%+rd%+10,ys%=ym%+rd%+10
xe%=xm%-rd%-10,ye%=ym%-rd%-10
BOX xs%,ys%,xe%,ye%      // position of the light
//                          source
```

```
lx=-1.5,ly=-1.5,lz=-0.25
//                          length of the light
//                          vectors
ll=SQR(lx^2+ly^2+lz^2) //   loop to get all points on
//                          the ball surface
COLOR 11
FOR i%=SUB(xm%,rd%) TO ADD(xm%,rd%)
  FOR j%=SUB(ym%,rd%) TO ADD(ym%,rd%)
    IF SQR((i%-xm%)^2+(j%-ym%)^2)<rd%
      vx%=SUB(i%,xm%),vy%=SUB(j%,ym%)
      vz%=SQR(rd%^2-vx%^2-vy%^2)
      vl=SQR(vx%^2+vy%^2+vz%^2)
      w=(lx*vx%+ly*vy%+lz*vz%)/(ll*vl)
      IF w+RND(1)*1.5-0.75>0
        PLOT i%,j%
      ENDIF
    ENDIF
  NEXT j%
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

Draws a red-framed box and inside it a ball which is illuminated from a light source in the upper left corner.

The GFA-BASIC command **PSET** is similar to **PLOT**. The difference is that **PSET** needs three parameters, where the third parameter is the colour of the graphic point.

The first two parameters specify - just like for **PLOT** - the X and Y coordinates of screen point.

Example:

```
SCREEN 16                 // EGA mode
DO
  MOUSE mx%,my%,mk%       // determine mouse position
    IF mk% & 1            // left mouse button?
 f%=RAND(1000) & 15       // colour
      PSET mx%,my%,f%     // set point
    ENDIF
LOOP UNTIL mk% & 2        // until the right mouse
//                           button is down
SCREEN 3
```

The program draws a graphic point of random colour when the left mouse button is pressed. The program is terminated by pressing the right mouse button.

The **DEFLINE** command determines how the appearance of the line drawn with **LINE, BOX, RBOX, CIRCLE, ELLIPSE** and **POLYLINE** commands. **DEFLINE** requires four parameters:

The first defines the line style. The following options are possible:

```
n = 0    line in background colour
n = 1    dashed line
n = 2    dashed line with short gaps
n = 3    dotted line
n = 4    dash-dot line
n = 5    dashed line with long gaps
n = 6    -..-..-..
```

The second parameter determines the thickness of the line in pixels. Only odd values are allowed for the line thickness.

The third and fourth parameters determine the line start and/or end symbols. The following options are available:

```
0 -> square
1 -> arrow
2 -> round
```

The parameters in front can be left out if the parameter separators, commas or spaces, are entered instead.

DEFLINE „11 sets the start and end symbols to arrow, but leaves the line style and width unchanged.

Example:

```
SCREEN 16                    // EGA mode
FOR i%=1 TO 6
  DEFLINE i%
  LINE 50,i%*50,200,i%*50
NEXT i%
DEFLINE 1,1,1,2
FOR i%=2 TO 12 STEP 2
  DEFLINE ,i%
  LINE 250,i%*24,400,i%*25
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

Draws lines with various styles and thickness.

The **LINE** command draws a line on the screen whose characteristics were set with **DEFLINE**. **LINE** requires four parameters, whereby the first two are the X and Y coordinates of the starting point and the last two the X and Y coordinates of the end point.

Example:

```
SCREEN 18                    // VGA mode
sx%=_X/2-_X/4                // start coordinates for X
ex%=_X/2+_X/4                // end coordinates for X
sy%=_Y/2-_Y/4                // start coordinates for Y
ey%=_Y/2+_Y/4                // end coordinates for Y
COLOR 4                      // red background
PBOX sx%,sy%,ex%,ey%
GRAPHMODE                    // XOR
COLOR 14                     // yellow
FOR i%=sx% TO ex%
  LINE i%,sy%,_X/2,_Y/2
  LINE i%,ey%,_X/2,_Y/2
NEXT i%
COLOR 11                     // green
FOR i%=sy% TO ey%
  LINE sx%,i%,_X/2,_Y/2
  LINE ex%,i%,_X/2,_Y/2
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

This example draws a moire pattern in yellow and green on red background.

The GFA-BASIC command **DRAW** is the most powerful graphic command for drawing of points and lines. Depending on its function, **DRAW** requires one, two, three, four or a multiple of two as parameters.

DRAW x,y corresponds to the **PLOT** command, i.e. it draws a point with the coordinates X and Y on the screen.

DRAW TO x,y draws a line between the point with the coordinates X and Y and the last previously defined point. It does not matter whether this point was set with **PLOT, LINE** or **DRAW**.

DRAW x1,y1 TO x2,y2 corresponds to the **LINE** command. **DRAW** allows for specification of additional coordinates with **TO**. In this was it is possible to draw connected lines for example.

DRAW exp enables implementation of commands which are similar to certain graphic commands in **LOGO** (turtle graphics) and the standard plotter language HPGL from Hewlett-Packard. In this way it is possible to move an imaginary pen around the screen which will draw relative to its position. The parameters for individual commands are always floating point numbers but can also be specified in strings.

The following commands are available:

FD n	ForwarD	moves the 'pen' n pixels 'forward'.
BK n	Backward	moves the 'pen' n pixels 'backward'.
SX x	Scale X	scales the 'pen movement' for FD
SY y	Scale Y	and BK by the factor specified in x and y. The scaling can be turned off with SX 0 and SY 0.
LT w	Left Turn	rotates the 'pen' by the angle in w (in degrees) to the left, RT w does the same to the right.
TT w	Turn To	turns the 'pen' to an absolute angle (in degrees). The following applies: w = 0: up or north w = 90: right or east w = 180: down or south w = 270: left or west
MA x,y	Move Absolute	moves the ''pen'to the absolute coordinates X and Y.
DA x,y	Draw Absolute	moves the 'pen' to the absolute coordinates X and Y and draws a line in current colour from the last point to point (x,y).
MR x,y	Move Relative	same as MA, but relative to the last position

DR x,y	Draw Relative	same as MR, but relative to the last position
CO n	Color	sets the colour n as character colour.
PU	Pen Up	moves the 'pen' up.
PD	Pen Down	sets the 'pen' down.

DRAW(i) is a function which returns the following values for depending on i:

i = 0	X position (floating point value)
i = 1	Y position (floating point value)
i = 2	angle in degrees (floating point value)
i = 3	scaling on the X axis (floating point value)
i = 4	scaling on the Y axis (floating point value)
i = 5	pen flag (-1 for PD and 0 for PU)

Example:

```
SCREEN 16              // EGA mode
msize%=1000000         // parameters for a fractal
wl%=-1000
wr%=250
wt%=625
wb%=-625
xl%=_X/2
yl%_Y/2
y%=0
//
REPEAT
  oi%=0,ox%=0
  loicy%=wt%+y%*(wb%-wt%)/yl%
  FOR x%=0 TO xl%-1
    loicx% = wl%+x%*(wr%-wl%)/xl%
    mandely%=loicy%,mandelx%=loicx%
    i%=0
    WHILE i% < 30
```

```
realnum%=mandelx%*mandelx%
imagnum%=mandely%*mandely%
      EXIT IF realnum%+imagnum% >= msize%
      mandely%=mandelx%*mandely%\250+loicy%
      mandelx%=(realnum%-imagnum%)\500+loicx%
      INC i%
    WEND
    DRAW CO oi% & 15
    IF ox%<x%-1
      DRAW ox%,y% TO x%-1,y%
    ELSE
      DRAW ox%,y%
    ENDIF
    ox%=x%,oi% = i%
  NEXT x%
  IF i%=oi%
    COLOR i% & 15
    DRAW ox%,y% TO x%-1,y%
  ENDIF
  y%++
UNTIL y%=yl%-1
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

This example draws a fractal picture in the upper left corner of the screen.

The **SETDRAW** enables default definitions of certain commands for the **DRAW exp** command.

SETDRAW x,y,w corresponds to the **DRAW "MA",x,y"TT",w** command.

Example:

```
SCREEN 16                   // EGA mode
FOR i%=0 TO 359
  SETDRAW 320,200,i%
  GRAPHMODE 6
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
  GRAPHMODE 1
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
NEXT i%
```

This example draws one small and one large rectangle which rotate around their axis.

The **BOX** command draws a box on the screen. The command requires four parameters. The first two specify the coordinates of the upper left corner and the last two the lower right corner. If **COLOR** is invoked before the **BOX** command the colour of the box can be changed.

Example:

```
SCREEN 16                   // EGA mode
x%=10,y%=10,b%=300,h%=150
COLOR 4
BOX x%,y%,x%+b%,y%+h%
p%=77,q%=2,r%=4
b%>>=1,h%>>=1
COLOR 14
xalt%=ADD(ADD(b%,b%),x%),yalt%=ADD(h%,y%)
FOR i%=p% TO 180*p% STEP p%
  xpos%=ADD(ADD(b%,b%*COSQ(q%*i%)),x%)
  ypos%=ADD(ADD(h%,h%*SINQ(r%*i%)),y%)
  LINE xalt%,yalt%,xpos%,ypos%
  xalt%=xpos%,yalt%=ypos%
```

```
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

This example draws a Lissajous figure inside a box.

The **RBOX** command has the same effect as **BOX**. In contrast to **BOX**, however, a rectangle with rounded corners is drawn instead.

The **PBOX** command draws - like the **BOX** command - a rectangle on the screen. The difference is in that the whole surface of the rectangle can be set with the **COLOR** command.

Example:

```
SCREEN 16                    // EGA mode
x%=10,y%=10,b%=300,h%=150
COLOR 1
PBOX x%,y%,x%+b%,y%+h%
p%=77,q%=2,r%=6
b%>>=1,h%>>=1
COLOR 14
xalt%=ADD(ADD(b%,b%),x%),yalt%=ADD(h%,y%)
FOR i%=p% TO 180*p% STEP p%
  xpos%=ADD(ADD(b%,b%*COSQ(q%*i%)),x%)
  ypos%=ADD(ADD(h%,h%*SINQ(r%*i%)),y%)
  LINE xalt%,yalt%,xpos%,ypos%
  xalt%=xpos%,yalt%=ypos%
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

This example draws a yellow Lissajous figure within a blue box.

To draw a **PBOX** with rounded corners use the **PRBOX** command.

The GFA-BASIC command **CIRCLE** draws a circle. Three parameters are needed for this. The first two specify the X and Y coordinates of the centre and the third the radius of the circle.

Example:

```
SCREEN 18                  // VGA resolution
COLOR 1
x%=_X/2,y%=_Y/2
PBOX SUB(x%,101),SUB(y%,101),ADD(x%,101),ADD(y%,101)
GRAPHMODE 24
FOR i%=1 TO 100
  COLOR i% & 15
  CIRCLE x%,y%,i%
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

The circles of various colours and with different radius are drawn in the middle of the screen within a blue rectangle.

Do pay attention when using the **CIRCLE** command that in EGA mode the screen points are rectangular and not square. **CIRCLE** therefore shows ellipses instead of circles when used in EGA mode. In CGA or "super" VGA modes the screen points are almost the same so that here a real circle is drawn.

To draw a filled circle use the GFA-BASIC command **PCIRCLE** . This command has the same parameters as **CIRCLE**, but is affected by the **COLOR** and **DEFFILL** commands.

The GFA-BASIC command **ELLIPSE** draws an ellipse. It requires four parameters. The first two specify the centre coordinates of the ellipse. The third parameter specifies the horizontal and fourth the vertical radius of the ellipse.

Example:

```
SCREEN 18                    // VGA mode
COLOR 1                      // blue
x%=_X/2,y%=_Y/2
PBOX SUB(x%,201),SUB(y%,101),ADD(x%,201),ADD(y%,101)
GRAPHMODE 3
FOR i%=1 TO 100
  IF i% & 2
    COLOR 14
  ELSE
    COLOR 11
  ENDIF
  ELLIPSE x%,y%,100+i%,i%
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

This program draws a number of growing ellipses on a blue background.

As with the **CIRCLE** command do note that the EGA resolution does not work with square screen points. This can cause the distortion of the ellipses.

To draw a filled ellipse use the GFA-BASIC command **PELLIPSE** . This command has the same parameters as **ELLIPSE,** but is affected by the **COLOR** and **DEFFILL** commands.

The GFA-BASIC command **CURVE** draws a Bezier curve and requires eight parameters. The first two parameters set the start of the curve. The last two parameters set the end of the Bezier curve. The third and fourth as well as the fifth and sixth parameters specify again an X and Y coordinate. If the parameters are x0,y0,x1,y1,x2,y2,x3 and y3, then is the curve in x0,y0 a tangent to the line from x0,y0 to x1,y1 and in x3,y3 a tangent to the line between x3,x3 and x2,y2. If the points in x0,y0,...,x3,y3 are viewed as the corners of a square, the curve lies fully within the square defined with these points.

The curve can also be viewed as a line between x0,y0 and x3,y3, which is pulled from the points x1,y1 and x2,y2.

Example:

```
SCREEN 18                        //      VGA mode
COLOR 4                          //      red
BOX 20,20,400,200                //      rectangle
LINE 20,20,400,200               //      diagonal
COLOR 11                         //      light blue
CURVE 20,20,310,20,110,200,400,200
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

This program draws a light blue Bezier curve in a rectangle with red borders. If you imagine a centred square inside the rectangle, the Bezier curve is symmetrically influenced by its upper right and lower left corners.

This accounts for the symmetrical deviation from the diagonal.

The GFA-BASIC command **POLYLINE** draws joined lines with a defined number of corners. The command requires as parameters the expression n, which specifies the number of corners, as well as two arrays, the first of which, x(), lists the X coordinates and the second, y(), the Y coordinates of the corner points. The first corner is specified in x(0),y(0), the last in x(n-1),y(n-1). The first and last corners will be connected automatically with each other. Optionally, a horizontal and/or vertical offset can be added to these coordinates.

Example:

```
SCREEN 16                // EGA mode
DIM x%(3),y%(3)
DATA 120,120,170,170,70,170,120,120
COLOR 4
BOX 65,115,175,175
FOR i%= 0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
COLOR 11
POLYLINE 4,x%(),y%()
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

In this example a light blue triangle is drawn in a rectangle with red edges on a black background.

The GFA-BASIC command **POLYFILL** draws a filled polygon with n corners. **POLYFILL** is to **POLYLINE** what **PBOX** is to **BOX** or **PELLIPSE** to **ELLIPSE**. It follows therefore that **POLYFILL** requires the same parameters as **POLYLINE**.

Example:

```
SCREEN 18              // VGA mode
DIM x%(3),y%(3)
DATA 120,120,170,170,70,170,120,120
COLOR 4
BOX 65,115,175,175
FOR i%= 0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
COLOR 11
DEFFILL 5
POLYFILL 4,x%(),y%() OFFSET -55,-55
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

In this example a light blue triangle is drawn with a fill pattern. By using **OFFSET** the triangle is moved left to the upper left corner of a rectangle with red edges.

The GFA-BASIC command **FILL** fills an arbitrary area. Three parameters are required for this. The first two specify the X and Y coordinates of the point, from which the fill should begin. The third parameter is optional. When specified, the fill is limited by points with the colour specified in the third parameter.

Example:

```
SCREEN 18                  // VGA mode
LINE 0,_Y-120,_X,_Y-120
FOR i%=1 TO 10
  BOX i%*20,_Y/2,i%*20+20-i%,_Y-120
  TEXT i%*20-4,_Y-105,i%
  DEFFILL ,2,i%
  FILL i%*20+1,_Y+1,1
NEXT i%
```

This program draws a straight line successive rectangles using different patterns and fills then with colour limitation.

The GFA-BASIC command **TEXT** displays a numeric or string expression as graphic text. The command requires three parameters. The first two specify the X and Y coordinates of where to print, and the third parameter the expression to print. The coordinates of where to print define the left edge of the base line of the first character to print.

Example:

```
SCREEN 18                     // VGA mode
a$="GFA-BASIC"                // expression to print
z%=0
DO
  COLOR 0                     // black
  z%++
  PBOX 0,0,_X,_X              // black background
  COLOR 14,4
  SELECT z%
  CASE 1
    FOR i%=1 TO _Y/2-30
      TEXT i%,i% a$
      TEXT _X-i%-72,i%,a$
      TEXT i%,_Y-i%,a$
      TEXT _X-i%-72,_Y-i%,a$
    NEXT i%
  CASE 2
```

```
    FOR i%=1 TO _Y/2-30
      TEXT i%,i% a$
      TEXT _X-i%-72,i%,a$
    NEXT i%
  CASE 3
    FOR i%=1 TO _Y/2-30
      TEXT i%,_Y-i%,a$
      TEXT _X-i%-72,_Y-i%,a$
    NEXT i%
  CASE 4
    FOR i%=1 TO _Y/2-30
      TEXT i%,i% a$
      TEXT i%,_Y-i%,a$
    NEXT i%
  CASE 5
    FOR i%=1 TO _Y/2-30
      TEXT _X-i%-72,i%,a$
      TEXT _X-i%-72,_Y-i%,a$
    NEXT i%
  CASE 6
    FOR i%=1 TO _Y/2-30
      TEXT i%,i% a$
      TEXT _X-i%-72,_Y-i%,a$
    NEXT i%
  CASE 7
    FOR i%=1 TO _Y/2-30
      TEXT _X-i%-72,i%,a$
      TEXT i%,_Y-i%,a$
    NEXT i%
  ENDSELECT
  COLOR 14,1
  TEXT 172,_Y/2," programmed by Frank Ostrowski "
  FOR j%=1 TO 300000
```

```
  NEXT j%
  IF z%>6
    z%=0
  ENDIF
UNTIL LEN(INKEY$)
SCREEN 3
```

This example displays the expression "GFA-BASIC" in various ways on the screen. The middle of the screen contains "programmed by Frank Ostrowski".

As in the text mode, the graphic mode can also limit the output of a graphic to a certain area. The GFA-BASIC command **CLIP** is used for this purpose. **CLIP** requires two or four parameters and can be used in four ways:

```
CLIP x,y,w,h[ OFFSET x0,y0]              or
CLIP x1,y1 TO x2,y2[ OFFSET x0,y0]       or
CLIP #n                                  or
CLIP OFFSET x0,y0                        or
CLIP OFF
```

CLIP x,y,w,h[OFFSET x0,y0] limits the graphic output to any rectangular area. By using the optional command **OFFSET x0,y0** the origins of the graphic output are set to the point with X coordinate x0 and the Y coordinate y0. x and y specify the X and Y coordinates for upper left corner of the clipping window. w (width) and h (height) specify the width and height of the clipping window.

For CLIP x1,y1 TO x2,y2 the x1 and y1 specify the upper left corner, x2 and y2 the lower right corner of the clipping rectangle.

CLIP #n enables the limitation of the output to the window with the number n (see below).

CLIP OFFSET x0,y0 specifies the origin for the output to the point with the coordinates x0,y0.

To turn off the clipping set with **CLIP** GFA-BASIC command **CLIP OFF** should be used.

Example:

```
SCREEN 18                 // VGA mode
DIM x%(3),y%(3)
DATA 120,120,170,170,70,170,120,120
COLOR 4
BOX 65,115,175,175
FOR i%= 0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
COLOR 11
DEFFILL 5
POLYFILL 4,x%(),y%() OFFSET -35,-35
REPEAT
UNTIL LEN(INKEY$)
COLOR 0
PBOX 0,0,_X,_Y
COLOR 4
BOX 65,115,175,175
CLIP 65,115 TO 175,175
POLYFILL 4,x%(),y%() OFFSET -35,-35
REPEAT
UNTIL LEN(INKEY$)
CLIPP OFF
POLYFILL 4,x%(),y%() OFFSET -35,-35
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

A filled triangle, which extends beyond the left and top edge of a red-framed box, is drawn first.

The program then waits for a keypress, erases the screen, redraws the rectangle and sets the clipping

rectangle with the same coordinates as the red-framed box. The filled triangle is drawn again with clipping on so that only the portions of the triangle which lie within the box are drawn. The program waits again for a keypress and removes the clipping. The triangle is the drawn again whereby it extends beyond the box.

The GFA-BASIC command **GET** saves a segment of the graphic screen in a string variable. The command requires five parameters. The first two define the X and Y coordinates of the upper left corner of the segment. The third and fourth parameters specify the X and Y coordinates of lower right corner of the segment. The fifth parameter is the string variable where the segment is to be saved.

Do note that when using **GET** the maximum size in one string does not exceed 32 KBytes. This can happen very frequently with VGA or higher resolutions. If needed one must work with more than one **GET** command.

Example:

```
SCREEN 18                   // VGA mode
GRAPHMODE 3
FOR i%=0 TO 15
  COLOR i%
BOX i%,i%,SUB(39,i%),SUB(39,i%)
NEXT i%
GET 0,0,39,39,a$
SCREEN 3
TCOLOR 4
PRINT LEN(a$)
REPEAT
UNTIL LEN(INKEY$)
```

The length of the character string a$, which contains the screen segment, is in this case 806 bytes. If the screen segment obtained with **GET** (VGA mode!!) is increased by 1/6 of the resolution, i.e.

```
GET 0,0,_X/2,_Y/3,a$
```

the length of a$ is already 25606 bytes. The whole screen must therefore be saved in six string variables.

The GFA-BASIC command **PUT** writes the screen segment saved with **GET** back to the screen. **PUT** requires three parameters, whereby the first two specify the X and Y coordinates where the saved screen segment is to appear. The third parameter defines the string variable where the segment should be read from.

By specifying a fourth, optional, parameter "mode" the way in which the saved bit pattern is to be moved to the screen is determined. "mode" assumes the same values as in **GRAPHMODE.** They are:

mode	rule	effect
1	replace	the current screen pattern in the area to write to is replaced by the segment bit pattern.
2	OR	the points which are set in both, the segment and the screen, are set.
3	XOR	only the points which are different in the screen and the segment are set.
4	AND	only the points which are set in both the screen and the segment are set.

Example:

```
SCREEN 18                // VGA mode
GRAPHMODE 3
FOR i%=0 TO 15
  COLOR i%
BOX i%,i%,SUB(39,i%),SUB(39,i%)
NEXT i%
GET 0,0,39,39,a$
v%=_X/3,w%=_Y/3,vv%=2,ww%=2
REPEAT
  ADD i%,4
```

```
    PUT _X/2+v%*SINQ(i%),_Y/3+w%*COSQ(i%),a$
    v%+=vv%,w%+=ww%
    IF v%<0 || v%>_X/3
      vv%=-vv%
    ENDIF
    IF w%<0 || w%>_Y/3
      ww%=-ww%
    ENDIF
  UNTIL LEN(INKEY$)
  SCREEN 3
```

This program draws first a rectangle out of multicoloured smaller rectangles. This is displayed in the upper left corner of the screen and read into variable a$. This segment is then written to various places on the screen from the **REPEAT...UNTIL** loop by using **PUT**.

The GFA-BASIC command **EMSGET** serves to save screen segments directly to EMS memory. This enables storage of segment greater than 32 KBytes. If there is no EMS, the storing can be redirected to the file, whose name and path are defined with **EDIR** . **EMSGET** has the same parameters as **GET**. In addition, it is possible to specify a name at the end of the **GET** command - up to 16 characters long and separated by a comma - under which the segment should be saved. If the EMS or other media does not have sufficient amount of memory, **EMSGET** produces an error message. It is therefore advisable to check if there is enough memory beforehand by using **EAVAIL**.

The complementary command to **EMSGET** is the GFA-BASIC command **EMSPUT**. It works like **PUT**, but it reads the segment directly from EMS memory or a file to the screen. **EMSPUT** requires the same parameters as **PUT**. However, there is one difference:

If a screen segment is saved to a string variable with a **GET** and then restored back to the screen with **PUT**, the string variable still contains the segment.

If an area of memory is written directly to EMS with **EMSGET** and then unloaded with **EMPUT**, the memory area is deleted from EMS (i.e. is released). To use **EMSGET** and **EMSPUT** like **GET** and **PUT**, the last parameter in **EMSGET** must be a name which **EMSPUT** (last optional parameter) can use to write the area out.

EMSPUT x,y:name does not release the relevant area in EMS memory. This happens only after **EKILL name**, which forces the freeing of the area.

Do note also that EMS operations **EMSGET** and **EMSPUT** are slower than **GET** and **PUT**. This is mainly because of the complicated management of EMS memory. As a demonstration this example written with **PUT** is replaced by the corresponding EMS command:

Example:

```
SCREEN 18                   // VGA mode
GRAPHMODE 3
FOR i%=0 TO 15
  COLOR i%
BOX i%,i%,SUB(39,i%),SUB(39,i%)
NEXT i%
EMSGET 0,0,39,39,"Segment"
v%=_X/3,w%=_Y/3,vv%=2,ww%=2
REPEAT
  ADD i%,4
  t1=_X/2+v%*SINQ(i%)
  t2=_Y/3+w%*COSQ(i%)
  EMSPUT t1,t2:"Segment"
  v%+=vv%,w%+=ww%
  IF v%<0 || v%>_X/3
    vv%=-vv%
```

```
    ENDIF
    IF w%<0 || w%>_Y/3
      ww%=-ww%
    ENDIF
  UNTIL LEN(INKEY$)
  SCREEN 3
```

3. The commands and functions for creation of a (pixel)graphic interface

GFA-BASIC provides three groups of resources for creation of a graphic user interface:

1. **Menu bars with pull-down menus**
2. **Alert boxes and pop-up menu**
3. **Windows**

The following chapters describe the creation and usage of such an interface.

3.1 Menu bars

The menu is a tool for program control. Analogous to a restaurant menu it contains various dishes (called menu entries) which can be selected. The effect of this selection consists, at the very least, of a branch to a particular part of the program. In order to achieve better program structure several menus should be used, each covering a related group of functions.

The GFA-BASIC commands **MENU** and **SYSCOL**

Let's assume you wish to write a program to create and use a data base. In such a case you would need four basic functions:

a) You must be able to edit the data.
b) You must be able to evaluate the data (display for example).
c) You must save the data to a medium and
d) must be able to load it back.

In the above example you should use two menus, the first menu to contain the entries "Load data" and "Save data" (functions c and d) and the second menu to contain the entries "Edit data" and "Display data" (functions a and b). The menus should be called "File" and "Edit" respectively.

Such menus appear in a program in a menu bar, which is located at the top of the (graphic) screen.

To create a menu bar in GFA-BASIC one requires a one-dimensional string array which contains the titles of individual menus, their entries and blank strings as separators between menus.

The menu bar is displayed in GFA-BASIC with the **MENU array$()** command, where array$() specifies the string array which contains the menus.

Example:

```
SCREEN 16                // EGA mode
build_menu()
//
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
//
//
PROCEDURE build_menu()
  DIM a$(10)
  a$(0)=" File "
  a$(1)="_Load data"
  a$(2)="_Save data"
  a$(3)=""
  a$(4)=" Edit "
  a$(5)="_Edit data"
  a$(6)="_Display data"
  a$(7)=""
  MENU a$()
  OPENW #0
RETURN
```

Since a graphic interface can only be used in the graphic mode the example program switches first to EGA mode. The **PROCEDURE** build_menu is invoked next to create the menu bar. This procedure declares a one-dimensional string array a$() which will, from now on, serve as the menu array. The elements in a$() are them initialised with menu titles and menu entries. Do note that for **OPTION BASE 0** a$(0) must contain the title of the first menu. For **OPTION BASE 1** this title is in a$(1). A blank string "" is used as a separator between the two menus as well as after the last menu.

The GFA-BASIC **MENU a$()** command then displays on the screen the menu bar with entries from the menu array a$(). The strings "File" and "Edit" are the titles of the two menus. The **OPENW #0** command serves to protect the menu bar from being overwritten (see below).

If the mouse driver has been loaded, then in addition to the menu bar the screen also contains the mouse pointer. You will also notice that the screen is black and the menu bar is grey. At this stage the individual menus cannot yet be opened. Before reaching this point the GFA-BASIC command **SYSCOL** should be used.

SYSCOL serves to set the colours in Menu bars, Pull-down menus, Pop-up menus, Windows and 3D effects. The command requires three parameters: **o, vc** and **hc**.

o defines the object which will be affected by the command. The following applies:

o = 0	Menu bar
o = 1	Pull-down menu
o = 2	Pop-up menu
o = 3	Windows
o = 4	3D effects

o = 5	Text in windows and pop-up menus
o = 6	Desktop
o = 7	Desktop fill pattern
o = 8	Alert boxes

vc defines the character colour and hc the background colour. The following applies for **CGA resolution**:

1. Palette		**2. Palette**	
vc or hc = 1	turquoise	vc or hc = 1	green
vc or hc = 2	violet	vc or hc = 2	red
vc or hc = 3	white	vc or hc = 3	yellow

for **EGA** or **VGA resolution**:

vc or hc = 0	black
vc or hc = 1	blue
vc or hc = 2	green
vc or hc = 3	turquoise (cyan)
vc or hc = 4	red
vc or hc = 5	purple (magenta)
vc or hc = 6	brown
vc or hc = 7	light grey
vc or hc = 8	dark grey
vc or hc = 9	light blue
vc or hc = 10	light green
vc or hc = 11	light turquoise
vc or hc = 12	light red
vc or hc = 13	light purple
vc or hc = 14	light yellow
vc or hc = 15	white

To display the menu titles in yellow on a blue menu bar in the above example, insert the command **SYSCOL 0,14,1** before **MENU a$()** in **PROCEDURE** build_menu:

Example:

```
      PROCEDURE build_menu()
        DIM a$(10)
        a$(0)=" File "
        a$(1)="_Load data"
        a$(2)="_Save data"
        a$(3)=""
        a$(4)=" Edit "
        a$(5)="_Edit data"
        a$(6)="_Display data"
        a$(7)=""
***     SYSCOL 0,14,1
        MENU a$()
        OPENW #0
      RETURN
```

As already explained the **MENU array$()** command only displays the menu bar with the menu entries on the screen. To "unfold" a menu, i.e. to activate it, one needs a method to manage the events in a menu bar. In general, there are three commands in GFA-BASIC for event management in menu bars, pop-up menus and windows:

PEEKEVENT	or
PEEK_EVENT	
GETEVENT	or
GET_EVENT	and
ON MENU	

PEEKEVENT or **PEEK_EVENT** tests only once if an event has occurred (for example menu entry selection, pop-up menu entry selection, keypress, mouse button press, window event). The program then continues.

GETEVENT or **GET_EVENT** waits half a second for an event to occur. The program then continues.

ON MENU waits like **GETEVENT** for an event to occur. If it does, **ON MENU** calls one of the subroutines defined with

> **ON MENU KEY GOSUB** procedurename
> **ON MENU BUTTON GOSUB** procedurename
> **ON MENU MESSAGE GOSUB** procedurename
> **ON MENU GOSUB** procedurename

The relevant subroutine branching must be defined before the **ON MENU** command!!

When the "corresponding" event occurs, an internal array is filled with event specific information for any of these three commands (a "corresponding event" is an event caused by one the specified actions). This action can be evaluated with the GFA-BASIC **MENU()** function.

MENU()

The structure of the event array **MENU()** is as follows:

MENU(1)=1	keyboard
MENU(4)	information about the pressed key. The low byte contains the ASCII code and the high byte the scan code of the pressed key.
MENU(5)	status of the special keys; KbShift
MENU(1)=2	a mouse click outside of the active window (Top Window)
MENU(7)	returns the number of the window, over which the mouse was situated when the mouse click occurred.
MENU(1)=3	mouse click inside the active window (Top Window)
MENU(1)=4	the close field of a window was activated.
MENU(1)=5	the minimum size field of a window was activated.
MENU(1)=6	the maximum size field of a window was activated.
MENU(1)=7	the arrow up field of a window was activated.
MENU(1)=8	the arrow down field of a window was activated.
MENU(1)=9	the arrow left field of a window was activated.
MENU(1)=10	the arrow right field of a window was activated.

MENU(1)=11	the area above the vertical scroll bar was activated; Page up.
MENU(1)=12	the area below the vertical scroll bar was activated; Page down.
MENU(1)=13	the area to the left of the horizontal scroll bar was activated; Page left.
MENU(1)=14	the area to the right of the horizontal scroll bar was activated; Page right.
MENU(1)=15	the vertical scroll bar was moved:
MENU(7)	position in the range from 0 to 1000.
MENU(1)=16	the horizontal scroll bar was moved:
MENU(7)	position in the range from 0 to 1000.
MENU(1)=17	the title field of a window was activated. If the window was moved then:
MENU(7)	new X coordinate,
MENU(8)	new Y coordinate of the upper left corner of the window.
MENU(1)=18	the size field of a window was activated. If the size of the window was changed then:
MENU(7)	new width,
MENU(8)	new height of the window.
MENU(1)=19	the info line of a window was clicked on.
MENU(1)=20	a menu bar entry or a pop-up menu entry was selected.
MENU(0)	returns the menu entry index or the number of the entry in the pop-up menu.
MENU(7)	returns the number of the menu (1,2,..).
MENU(8)	returns the number of the entry (1,2...) in menu 1,2,...
MENU(1)=21	a rectangular segment of the window must be redrawn; Redraw Message:
MENU(7)	returns the left X coordinate of the rectangle,
MENU(8)	returns the upper Y coordinate of the rectangle,
MENU(9)	returns the width of the rectangle,
MENU(10)	returns the width of the rectangle.

The following is always:

MENU(2)	X coordinate of the mouse,
MENU(3)	Y coordinate of the mouse.
MENU(4)	the status of the mouse buttons, as follows:
MENU(4)=0	no mouse buttons were clicked,
MENU(4)=1	the left mouse button was clicked,
MENU(4)=2	the right mouse button was clicked.

To activate a menu in the above example the event monitoring must be turned on first. To do this the lines

```
REPEAT
UNTIL LEN(INKEY$)
```

must be deleted and the following

```
DO
  GETEVENT
  e%=MENU(1)
UNTIL e%=20
```

inserted instead.

Example:

```
         SCREEN 16                  // EGA mode
         build_menu()
         //
   ***   DO
   ***     GETEVENT
           e%=MENU(1)
   ***   UNTIL e%=20
         SCREEN 3
         //
         PROCEDURE build_menu()
           DIM a$(10)
```

```
                  a$(0)=" File "
                  a$(1)="_Load data"
                  a$(2)="_Save data"
                  a$(3)=""
                  a$(4)=" Edit "
                  a$(5)="_Edit data"
                  a$(6)="_Display data"
                  a$(7)=""
***               SYSCOL 0,14,1
                  MENU a$()
                  OPENW #0
                RETURN
```

As you can see from the event array layout, this loop waits for an entry to be selected from a menu bar (**MENU(1)=20**).

If you now run the program, you can open a menu, move the mouse pointer to the relevant entry and click the left mouse button. Optionally, you can press the F1 function key. This will open the first menu. You can then use the cursor keys to move to the menu you wish to open. There are three ways to select a menu entry:

1. move the mouse pointer to the corresponding entry and press the left mouse button.
2. select an entry with the cursor keys and then press the Return key.
3. when building the menu entry array a$() define the so-called "hotkeys" for various entries by prefixing the specific letters with an underdash "_". You can then select the relevant entry in an active menu by pressing the key with this letter.

If possible (i.e. if the mouse driver is installed) you should now try all of these alternatives in the above example.

You will notice that in an active menu (unfolded pull-down menu) the pull-down menu is shown in grey with black entries. If you wish to change

the colours of this pull-down menu, for example yellow writing of red background, you must insert the line

```
SYSCOL 1,14,4
```

after the **MENUa$()** command in the **PROCEDURE** build_menu.

Example:

```
PROCEDURE build_menu()
  DIM a$(10)
  a$(0)=" FILE "
  a$(1)="_Load data"
  a$(2)="_Save data"
  a$(3)=""
  a$(4)=" Edit "
  a$(5)="_Edit data"
  a$(6)="_Display data"
  a$(7)=""
  SYSCOL 0,14,1
  MENU a$()
***  SYSCOL 1,14,4
  OPENW #0
RETURN
```

To react accordingly to the selected entry a **PROCEDURE** is needed to do the evaluation. This **PROCEDURE** should then be jumped to from the event monitoring in the **DO...UNTIL** loop (at the start of the program).

In our example **PROCEDURE** the evaluation procedure will be called evaluate_menu and it looks as follows:

Example:

```
PROCEDURE evaluate_menu()
  LOCAL ee_%
  ee_%=MENU(0)
  SWITCH ee_%
  CASE 1
    PRINT a$(ee_%)      // jumps to the corresponding
    //                     subroutine
```

```
    CASE 2
      PRINT a$(ee_%)       // jumps to the corresponding
      //                      subroutine
    CASE 5
      PRINT a$(ee_%)       // jumps to the corresponding
      //                      subroutine
    CASE 6
      PRINT a$(ee_%)       // jumps to the corresponding
      //                      subroutine
    ENDSWITCH
  RETURN
```

The evaluation procedure should be executed when a menu entry is selected. In this case **MENU(0)** in the menu array a$() contains the index of the selected entry. This index is moved to the local variable ee_% in the **PROCEDURE** evaluate_menu. The **SWITCH...CASE** conditional statement then reacts to the selected entry accordingly (in most cases with a jump to the relevant subroutine).

To jump to the **PROCEDURE** evaluate_menu the event monitoring:

```
DO
  GETEVENT
  e%=MENU(1)
UNTIL e%=20
```

must be modified with one of the following two alternatives.

Alternative 1:

```
DO
  GETEVENT
  e%=MENU(1)
  IF e%=20
    evaluate_menu()
  ENDIF
LOOP
```

Alternative 2:

```
ON MENU GOSUB evaluate_menu
DO
  ON MENU
LOOP
```

You will obviously recognise, that in the Alternative 1 the conditional statement is considerably clearer than in the Alternative 2. Which of the two alternatives you decide to use will depend on your programming style.

We have chosen the first alternative, since here the conditions for the jump are already apparent in the event monitoring.

If you run the modified program you'll notice that there is no option to terminate the program in a controlled fashion. A menu entry to terminate the program is missing. This entry should normally be the last entry in the first menu. To make it even more obvious it can be separated from the remaining entries with a separator line. After all of these changes (**PROCEDURE**s build_menu() and evaluate_menu() as well as the event monitoring were all modified!!) the example program appears as follows:

Example:

```
     SCREEN 16                           // EGA mode
     build_menu()
     //
     DO
       GETEVENT
       e%=MENU(1)
***    EXIT IF e%=20 && MENU(0)=4        // EXIT was selected
       IF e%=20
         evaluate_menu()
       ENDIF
     LOOP
     SCREEN 3
     //
     //
```

```
PROCEDURE build_menu()
  DIM a$(10)
  a$(0)=" File "
  a$(1)="_Load data"
  a$(2)="_Save data"
  a$(3)="----------------"
  a$(4)=" E_xit "
  a$(5)=""
  a$(6)=" Edit "
  a$(7)="_Edit data"
  a$(8)="_Display data"
  a$(9)=""
  SYSCOL 0,14,1
  MENU a$()
  SYSCOL 1,14,4
  OPENW #0
RETURN
//
PROCEDURE evaluate_menu()
  LOCAL ee_%
  ee_%=MENU(0)
  SWITCH ee_%
  CASE 1
    PRINT a$(ee_%)          // jumps to the corres-
    //                         ponding subroutine
  CASE 2
    PRINT a$(ee_%)          // jumps to the corres-
    //                         ponding subroutine
  CASE 7
    PRINT a$(ee_%)          // jumps to the corres-
    //                         ponding subroutine
  CASE 8
    PRINT a$(ee_%)          // jumps to the corres-
    //                         ponding subroutine
  ENDSWITCH
RETURN
```

If you've chosen the second alternative, you must check for program exit in the **PROCEDURE** evaluate_menu. The program then appears as follows:

Example:

```
SCREEN 16                    // EGA mode
build_menu()
//
ON MENU GOSUB evaluate_menu()
DO
  ON MENU
LOOP
//
//
PROCEDURE build_menu()
  DIM a$(10)
  a$(0)=" File "
  a$(1)="_Load data"
  a$(2)="_Save data"
  a$(3)="----------------"
  a$(4)=" E_xit "
  a$(5)=""
  a$(6)=" Edit "
  a$(7)="_Edit data"
  a$(8)="_Display data"
  a$(9)=""
  SYSCOL 0,14,1
  MENU a$()
  SYSCOL 1,14,4
  OPENW #0
RETURN
//
PROCEDURE evaluate_menu()
  LOCAL ee_%
  ee_%=MENU(0)
  SWITCH ee_%
```

```
          CASE 1
            PRINT a$(ee_%)                 // jumps to the corres-
            //                                ponding subroutine
          CASE 2
            PRINT a$(ee_%)                 // jumps to the corres-
            //                                ponding subroutine
          CASE 4
***         EDIT                           // program EXIT
          CASE 7
            PRINT a$(ee_%)                 // jumps to the corres-
            //                                ponding subroutine
          CASE 8
            PRINT a$(ee_%)                 // jumps to the corres-
            //                                ponding subroutine
          ENDSWITCH
        RETURN
```

3.2 Alert boxes and Pop-up menus

Alert boxes and Pop-up menus are tools for a dialogue between a program and the user. Basically, the alert boxes serve to provide information about a particular program state, while the pop-up menus only serve to perform a branch. This difference should be apparent in the following example:

Let's assume you've protected the data base in the above example from unauthorised access with a password. When the menu entries "Load data" or "Display data" are selected from the menu bar a request for the password should follow. This is what an alert box is for.

3.2.1 ALERT

An alert box is invoked from GFA-BASIC with the **ALERT** command. This command requires five parameters: **i,a$,j,b$,k.**

The first parameter, i, determines the appearance of the alert box, since an alert box can have a symbol in its upper left corner. The goal of this symbol is to graphically represent the purpose of the alert message. The following applies:

i	Symbol
0	no symbol
1	exclamation mark
2	question mark
3	stop sign

The second parameter, a$, contains the alert message. It can be up to three lines long. To start a new line, terminate the previous line with a vertical bar character "|":

```
a$ = "This data base is protected|"
a$ = a$+"Enter password"
```

In this example the alert message is composed of two lines.

The fourth parameter, b$, contains the options allowed by the alert box. Up to five alternatives are possible. The individual alternatives are separated by a vertical bar "|":

```
b$="Input|Return"
```

Two alternatives are envisioned here.

The third parameter, j, determines which of the alternatives is the default. The pressing the Return key then selects this default alternative.

The fifth parameter contains after the selection of an alternative its number.

In the example program the alert box is invoked from the subroutines branched to from the **PROCEDURE** evaluate_menu().

Example:

```
      SCREEN 16                          // EGA mode
      build_menu()
      //
***   code$="xy42z42"                    // the password
      //
      DO
        GETEVENT
        e%=MENU(1)
        EXIT IF e%=20 && MENU(0)=4       // EXIT was selected
        IF e%=20
          evaluate_menu()
        ENDIF
      LOOP
      SCREEN 3
      //
      //
      PROCEDURE build_menu()
        DIM a$(10)
        a$(0)=" File "
        a$(1)="_Load data"
```

```
  a$(2)="_Save data"
  a$(3)="----------------"
  a$(4)=" E_xit "
  a$(5)=""
  a$(6)=" Edit "
  a$(7)="_Edit data"
  a$(8)="_Display data"
  a$(9)=""
  SYSCOL 0,14,1
  MENU a$()
  SYSCOL 1,14,4
  OPENW #0
RETURN
//
PROCEDURE evaluate_menu()
  LOCAL ee_%
  ee_%=MENU(0)
  SWITCH ee_%
  CASE 1
    load_data()
    //
  CASE 2
    PRINT a$(ee_%)          // jumps to the corres-
    //                         ponding subroutine
  CASE 7
    PRINT a$(ee_%)          // jumps to the corres-
    //                         ponding subroutine
  CASE 8
    display_data()
    //
  ENDSWITCH
RETURN
//
```

***

```
PROCEDURE load_data()
  LOCAL a_$,b_$,c_$,dd_%,code_!
  //
  a_$="This data base is protected|"
```

```
               a_$=a_$+"Enter password"
               b_$="Input|Return"
               //
               DO
***              ALERT 1,a_$,1,b_$,dd_%
                 EXIT IF dd_%=2
                 INPUT"Password: ";c_$
                 IF c_$=code$
                   code_!=TRUE
                 ENDIF
               LOOP UNTIL code_!=TRUE
               //
               IF code_!=TRUE
                 //
                 //                           program continuation
                 //
               ENDIF
               //
             RETURN
             //
***          PROCEDURE display_data()
               LOCAL a_$,b_$,c_$,dd_%,code_!
               //
               a_$="This data base is protected|"
               a_$=a_$+"Enter password"
               b_$="Input|Return"
               //
               DO
***              ALERT 1,a_$,1,b_$,dd_%
                 EXIT IF dd_%=2
                 INPUT"Password: ";c_$
                 IF c_$=code$
                  code_!=TRUE
                 ENDIF
               LOOP UNTIL code_!=TRUE
               //
```

```
IF code_!=TRUE
  //
  //                          program continuation
  //
ENDIF
//
RETURN
```

The example program has been expanded to include the subroutines load_data() and display_data(). Both subroutines invoke an alert box, which performs the password inquiry. This inquiry is performed from a loop until a correct password is entered or the alternative "Return" is selected. The **IF...ENDIF** conditional statement after the **LOOP...** assures that the relevant subroutine is executed only after the correct password has been supplied.

And now for some cosmetic changes. When you run the program you will notice that the alert box is displayed in grey. To modify this colour somewhat the **SYSCOL** command is used again, this time with the value 8 as the first parameter.

SYSCOL 8,4,14 causes the text in the alert box to appear as red writing on yellow background. If you wish to try this out insert this line before the **DO...LOOP** loop:

Example:

```
PROCEDURE load_data()
  LOCAL a_$,b_$,c_$,dd_%,code_!
  //
  a_$="This data base is protected|"
  a_$=a_$+"Enter password"
  b_$="Input|Return"
  //
***   SYSCOL 8,4,14
  //
  DO
    ALERT 1,a_$,1,b_$,dd_%
```

```
      EXIT IF dd_%=2
      INPUT"Password: ";c_$
      IF c_$=code$
      code_!=TRUE
      ENDIF
    LOOP UNTIL code_!=TRUE
    //
    IF code_!=TRUE
      //
      //                        program continuation
      //
    ENDIF
    //
  RETURN
```

Let's assume you've divided your data base into several segments:

The first part contains the records which start with the letters A-F, the second G-J, the third with K-N, the fourth with O-R, the fifth with S and T and the sixth with U-Z. After entering the correct password you will offer to load one of these six records. You can use the GFA-BASIC **POPUP** function for this purpose:

3.2.2 POPUP

The **POPUP** function requires four parameters: **a$,x,y,i.**

The first parameter, a$, takes the entries of the pop-up menu. The first entry is reserved for the title of the pop-up menu. It is not selectable. The following entries are selectable and define the options in the pop-up menus. The various entries must be separated from each other by a vertical

bar "|". When an underdash "_" is used a prefix it defines the following character as a "Hotkey" (see above). For example you can write the following

```
a$="Data segment|_A-F|_G-J|_K-N|_O-R|_S,T|_U-Z"
```

The second (x) and third (y) parameters specifies the X and Y coordinates (in pixels) of the upper left corner of the pop-up menu.

The fourth parameter, i, is a control parameter. The following applies:

i = 0 The position specified with X and Y is relative to the upper left corner of the pop-up menu.

i = 1 The position specified with X and Y is relative to the middle of the pop-up menu.

i = 2 x,y is ignored; the pop-up menu is centred on the screen.

There are to ways of invoking a pop-up menu:

~POPUP(a$,x,y,i) or
a%=POPUP(a$,x,y,i)

Which of the two alternatives you choose depends on how the selected entry should be determined. The **POPUP** function fills in both cases the **MENU()** event array.

If you use the alternative

~POPUP(a$,x,y,i),

you must determine the selected entry from **MENU(1)=20** and **MENU(0)**.

But if you decide to use the alternative

a%=POPUP(a$,x,y,i),

a% immediately contains the number of the selected entry.

Which way you decided to go will in the end depend on the structure of your programs. If you need a general **PROCEDURE** for evaluation of both menu and pop-up events you should use the first alternative, otherwise you can use the second.

We will use the second alternative for out example program. The pop-up menu should appear only after the correct password has been entered. The pop-up menu must therefore be used in the **PROCEDURE**s load_data() and display_data() inside the **IF...ENDIF** conditional statement after the **DO...LOOP** loop. The evaluation of the selected entry should then occur in a **SWITCH...CASE** or **SELECT...CASE** conditional statement. The relevant changes are here shown only for the **PROCEDURE** load_data().

Example:

```
PROCEDURE load_data()
  LOCAL a_$,b_$,c_$,dd_%,code_!
  //
  a_$="This data base is protected|"
  a_$=a_$+"Enter password"
  b_$="Input|Return"
  //
  SYSCOL 8,4,14
  //
  DO
    ALERT 1,a_$,1,b_$,dd_%
    EXIT IF dd_%=2
    INPUT"Password: ";c_$
    IF c_$=code$
      code_!=TRUE
    ENDIF
  LOOP UNTIL code_!=TRUE
  //
```

```
          IF code_!=TRUE
            //
            a_$="Data segment|_A-F|_G-J|_K-N|_O-R|_S,T|_U-Z"
***         dd_%=POPUP(a_$,10,20,1)
            //
            SELECT dd_%
            CASE 1              /* A-F
              //                   load A-F
            CASE 2              /* G-J
              //                   load G-J
            CASE 3              /* K-N
              //                   load K-N
            CASE 4              /* O-R
              //                   load O-R
            CASE 5              /* S,T
              //                   load S,T
            CASE 6              /* U-Z
              //                   load U-Z
            OTHERWISE
              ALERT 3,"No segment was selec-
              ted",1,"Return",dd_%
            ENDSELECT
            //
          ENDIF
          //
        RETURN
```

In closing, another cosmetic change. To give the pop-up menus some colour the **SYSCOL** command can be used once more. **SYSCOL 5,4,14** causes the pop-up menu entries to be shown in red writing on yellow background. The command must be inserted before invoking the pop-up menu:

Example:

```
PROCEDURE load_data()
  LOCAL a_$,b_$,c_$,dd_%,code_!
  //
  a_$="This data base is protected|"
  a_$=a_$+"Enter password"
  b_$="Input|Return"
  //
  SYSCOL 8,4,14
  //
  DO
    ALERT 1,a_$,1,b_$,dd_%
    EXIT IF dd_%=2
    INPUT"Password: ";c_$
    IF c_$=code$
      code_!=TRUE
    ENDIF
  LOOP UNTIL code_!=TRUE
  //
  IF code_!=TRUE
    //
    a_$="Data segment|_A-F|_G-J|_K-N|_O-R|_S,T|_U-Z"
    //
    SYSCOL 5,4,14
    //
    dd_%=POPUP(a_$,10,20,1)
    //
    SELECT dd_%
    CASE 1              /* A-F
      //                      load A-F
    CASE 2              /* G-J
      //                      load G-J
    CASE 3              /* K-N
      //                      load K-N
    CASE 4              /* O-R
      //                      load O-R
    CASE 5              /* S,T
      //                      load S,T
```

```
    CASE 6              /* U-Z
      //                         load U-Z
    OTHERWISE
      ALERT 3,"No segment was
selected",1,"Return",dd_%
    ENDSELECT
    //
  ENDIF
  //
RETURN
```

3.3 Windows

One of the nicest features of GFA-BASIC is the creation and usage of (graphic) windows. Such a window can be created in graphic mode with the command

OPENW

This command requires six parameters: **nr,x,y,w,h,conf.**

The first parameter, nr, specifies which window should be opened. In GFA-BASIC for the MS-DOS operating system up to five windows can be opened, out of which only four can be manipulated further.

The window 0 is the Desktop. You must always open this window if you wish to use a menu bar and/or wish to change the screen background colour. The window 0 is opened with the GFA-BASIC command

OPENW #0

For example, to paint the background red and then write on it in yellow, you would use the following:

Example:

```
SCREEN 16               // EGA mode
COLOR 14,4
SYSCOL 6,14,4
OPENW #0
CLEARW #0
PRINT "Yellow on red"
```

The **COLOR** command in this example affects the **PRINT** output so that the writing is really yellow on red; the **SYSCOL** assignment affects window #0. If you leave the **COLOR** command out, the Desktop is still painted red, however, the **PRINT** command will produce yellow on blue background.

If you use a menu bar in your program, the opening of window #0 assures that the menu bar is not overwritten or, in case of a graphic command, "painted over".

Example:

```
SCREEN 16                  // EGA mode
DIM a$(2)
a$(0)="Title"
a$(1)="Quit"
a$(2)=""
MENU a$()
SYSCOL 6,14,4
OPENW #0
CLEARW #0
LINE 0,0,_X,_Y
COLOR 1,14
PRINT "The menu bar cannot";
PRINT "be overwritten"
DO
  GET_EVENT
UNTIL MENU(1)=20 && MENU(0)=1
EDIT
```

This means that all text and/or graphic output is clipped to the area covered by window #0.

The remaining four windows are numbered 1 to 4. These windows require additional parameters in the **OPENW** statements.

The parameters 2 and 3 specify the X and Y coordinates of the upper left corner of the window nr. The fourth parameter specifies the width and the fifth the height of the window. The sixth parameter is a word (2 bytes) whose bits describe the window attributes. Before we explore this parameter in detail we'll open a window in the following example.

Example:

```
SCREEN 16                  // EGA mode
OPENW #1,10,10,300,200,-1
REPEAT
UNTIL LEN(INKEY$)
EDIT
```

This example creates the window #1, 300 pixels wide and 200 pixels high, at position 10,10 (upper left corner). The -1 as the sixth parameter means that all bits are set in the configuration word. The window is therefore fully configured.

A rectangle with a start character appears in the upper left corner. This rectangle is the close field (Closer) for the window. To the right of the close field is a dotted bar (Mover), which - is so desired - can contain the title of the window. This bar is needed to move the window. To the right below the title bar are two more rectangles. The one with the arrow pointing down is the so-called Minimizer, and the one with the arrow pointing up the so-called Maximizer.

The functions of these four rectangles are as follows

(1) by clicking on the Closer the window is closed,

(2) by clicking on the Mover the window can be moved,

(3) by clicking on the Minimizer the window is reduced to the size of an Icon and

(4) by clicking on the Maximizer the window is enlarged to the size of the Desktop.

When these buttons are used your program must perform the required actions. How this is done will be described below.

Underneath the four described rectangles is another rectangle, the so-called window Info line. It is intended to contain additional information below the window title.

A rectangle with a diamond symbol is located in the lower right corner of the window. This is the resize field called the Sizer. By clicking on the Sizer and holding down the mouse button, the window can be expanded or shrunk when the lower right corner is moved in or out. Here too, your program must monitor these changes (see below).

The right side of the window underneath the Info line contains three rectangles. They relate to the vertical movements of the window contents. Clicking on the field with the triangle pointing up, causes the window contents to be moved one "unit" down. Conversely, clicking on the field with the triangle pointing down, moves the window contents one "unit" up. The "unit" is based on the relation between the actual vertical span of the window contents and the total maximum vertical span of window contents.

The dotted area is the area, within which the Slider can be moved vertically. If you open a window, as in the above example, the vertical Slider is located at the top of the slider area. It is shown as grey, white-framed rectangle.

The bottom left side of the window also contains three rectangles. They relate to the horizontal movements of the window contents.

As with the vertical sliders, a click on the field with the triangle pointing left, the window contents are moved one "until" to the right. Clicking on the field with the triangle pointing right, moves the window contents to the left. The "unit" is based on the relation between the actual horizontal span of the window contents and the total maximum horizontal span of window contents. The dotted area is the area, within which the Slider can be moved horizontally. If you open a window, as in the above example, the horizontal Slider is located at the left edge of the slider area.

Putting theory to practice

To demonstrate how window elements react to mouse actions a program must be written using event monitoring. We have chosen an endless loop for this task. The program can only be terminated by pressing the Control + Cancel.

Example:

```
SCREEN 16                    // EGA mode
OPENW #1,10,10,300,200,-1
DO
  GET_EVENT
LOOP
EDIT
```

Position the mouse pointer on the Closer and press the left mouse button. As a reaction to this event the button is inverted. It gives the impression of the button being pushed down. The same goes for the Mini- and Maximizer.

Move the mouse pointer to the Title bar and hold the left mouse button down. The mouse symbol changes into a hand. Now move the mouse while still holding the button down. You'll notice a rectangle the size of the window. This rectangle is used to move the window.

Move then the mouse pointer to the Sizer and hold the left mouse button down. If you now move the mouse, a rectangle appears whose upper left corner is at the same position as the upper left corner of the window, and whose lower right corner is the same as the mouse pointer position. This rectangle represents the new window area.

Move the mouse to the vertical or horizontal Slider and hold the left mouse button down. The mouse pointer changes into a hand. If you now move the mouse up and down or left and right the Slider symbol is drawn at the new location.

GFA-BASIC provides special routines for some of these events, the rest are supported by the combination thereof.

The GFA-BASIC commands

TITLEW
INFOW
CLOSEW
MOVEW
FULLW
SIZEW

TITLEW defines the window title in the title bar. The command requires two parameters: the number of the window which will get the new title (1,2,3,4) and, separated by a comma, a string which contains the window title. Modify the example program as follows:

Example:

```
SCREEN 16                // EGA mode
TITLEW #1," Window 1 "
OPENW #1,10,10,300,200,-1
DO
  GET_EVENT
LOOP
EDIT
```

When you run the program you'll see the string " Window 1 " appearing centred in the title bar of the window.

INFOW places a string on the Info line of the window. This command also requires two parameters, the window number and the string which is to appear on the Info line:

Example:

```
SCREEN 16                // EGA mode
TITLEW #1," Window 1 "
INFOW #1,"Info line"
OPENW #1,10,10,300,200,-1
DO
```

```
  GET_EVENT
LOOP
EDIT
```

When you run the program, the string "Info line" appears left justified on the Info line.

The only parameter for the **CLOSEW** is the number of the window (1,2,3 or 4) which is to be closed. This command should be used in a program as the reaction to the close event. The close event is clicking on the Closer. This event is obtained with the **GET_EVENT** and it means that in the event array **MENU()** (refer to "Menu bar" paragraph in this chapter) **MENU(1)** contains the value 4. The event evaluation and the relevant reaction are then included in the **DO ... LOOP** loop.

To prepare for the discussion of the upcoming commands we will preform the event evaluation with a **SWITCH...CASE** conditional statement:

Example:

```
        SCREEN 16                     // EGA mode
        TITLEW #1," Window 1 "
        INFOW #1,"Info line"
        OPENW #1,10,10,300,200,-1
        DO
          GET_EVENT
***       e%=MENU(1)
***       SWITCH e%
***       CASE 4                      // Closer event
***         CLOSEW #1
***       ENDSWITCH
        LOOP
        EDIT
```

The window will now be closed whenever the mouse pointer is moved to the Closer field and the left mouse button is pressed.

When you work with several windows it's likely that as soon as one window is closed the parts of the windows underneath it must be shown. GFA-BASIC reacts by redrawing the windows itself - but not the window contents. Your program must inquire about a so-called Redraw message, which writes the value 21 to **MENU(1)**. If there is such a message you must redraw the areas now visible by using a special routine. More about this later.

When you don't want to close a window but just wish to erase its contents, you should use the GFA-BASIC command **CLEARW** . **CLEARW** requires only one parameter, the number of the window (1, 2, 3 or 4) which is to be wiped clean.

MOVEW moves a window about the screen. The command requires three parameters: the number of the window and two values to specify the X and Y coordinates for the new window position. In order to move a window the relevant event must occur first, i.e. clicking on the move bar and moving of the window. This event stores the value 17 in **MENU(1)** and returns in **MENU(7)** the new X and in **MENU(8)** the new Y position of the upper left corner of the window.

Example:

```
SCREEN 16                // EGA mode
TITLEW #1," Window 1 "
INFOW #1,"Info line"
OPENW #1,10,10,300,200,-1
DO
  GET_EVENT
  e%=MENU(1)
  SWITCH e%
  CASE 4                 // Closer event
    CLOSEW #1
  CASE 17                // Mover event
```

***

```
***           MOVEW #1,MENU(7),MENU(8)
            ENDSWITCH
          LOOP
          EDIT
```

If you now use the Mover you'll see that the window is moved about the screen. To demonstrate the events, after a **MOVEW** for example, a small picture will be drawn in the window. The program is thus extended to include a **PROCEDURE** to perform the drawing. This **PROCEDURE** is called immediately after **OPENW**. The Info line is also changed.

Example:

```
          SCREEN 16                    // EGA mode
          TITLEW#1," Window 1"
***       INFOW #1,"Lissajous figure"
          OPENW#1,10,10,300,200,-1
***       l_figur()
          DO
            GET_EVENT
            e%=MENU(1)
            SWITCH e%
            CASE 4                     // Closer event
              CLOSEW #1
            CASE 17                    // Mover event
              MOVEW #1,MENU(7),MENU(8)
            ENDSWITCH
          LOOP
          EDIT
          //
***       PROCEDURE l_figur()
            LOCAL i_%,x_%,y_%,w_%,h_%
            LOCAL p_%,q_%,r_%
            LOCAL xa_%,ya_%,xp_%,yp_%
            p_%=77,q_%=2,r_%=3
            x_%=0,y_%=0
            w_%=_X,h_%=_Y
            w_%>>=1,h_%>>=1
```

```
    xa_%=ADD(ADD(w_%,w_%),x_%)
    ya_%=ADD(h_%,y_%)
    FOR i_%=p_% TO 180*p_% STEP p_%
      xp_%=ADD(ADD(w_%,w_%*COSQ(q_%*i_%)),x_%)
      yp_%=ADD(ADD(h_%,h_%*SINQ(r_%*i_%)),y_%)
      LINE xa_%,ya_%,xp_%,yp_%
      xa_%=xp_%,ya_%=yp_%
    NEXT i_%
  RETURN
```

When you run the program a figure using the Lissajous algorithm is drawn in the window.

If you now move the window about the screen with the mouse you'll observe the following:

As long as the window, in its full size, does not extend beyond the total screen area, both the window and its contents are redrawn after each **MOVEW**.

If the window extends beyond the screen edge and is redrawn by using **MOVEW**, only the window, and not the screen contents, are redrawn. To correct this you must inquire for a Redraw event (**MENU(1)=21**) and have your program react correspondingly. How this is done is explained further down.

FULLW performs the expansion of the window to its maximum size. This is the same as the Desktops size, i.e. the same as for window 0. If the menu bar is displayed, this area covers the whole screen below the menu bar (window 0 must protect the menu bar), otherwise (no menu bar) it covers the total screen area.

FULLW requires only one parameter, the number of the window (1,2,3 or 4). The event to extend the window to its maximum size is clicking on the Maximizer. This event stores the value 6 in **MENU(1)**. The relevant event monitoring as well as the program reaction to it are performed with the **SWITCH...CASE** conditional statement:

Example:

```
SCREEN 16                    // EGA mode
TITLEW#1," Window 1"
INFOW #1,"Lissajous figure"
OPENW#1,10,10,300,200,-1
l_figur()
DO
  GET_EVENT
  e%=MENU(1)
  SWITCH e%
  CASE 4                     // Closer event
    CLOSEW #1
*** CASE 6                   // Maximizer event
***   FULLW #1
  CASE 17                    // Mover event
    MOVEW #1,MENU(7),MENU(8)
  ENDSWITCH
LOOP
EDIT
//
PROCEDURE l_figur()
  LOCAL i_%,x_%,y_%,w_%,h_%
  LOCAL p_%,q_%,r_%
  LOCAL xa_%,ya_%,xp_%,yp_%
  p_%=77,q_%=2,r_%=3
  x_%=0,y_%=0
  w_%=_X,h_%=_Y
  w_%>>=1,h_%>>=1
  xa_%=ADD(ADD(w_%,w_%),x_%)
  ya_%=ADD(h_%,y_%)
  FOR i_%=p_% TO 180*p_% STEP p_%
    xp_%=ADD(ADD(w_%,w_%*COSQ(q_%*i_%)),x_%)
```

```
    yp_%=ADD(ADD(h_%,h_%*SINQ(r_%*i_%)),y_%)
    LINE xa_%,ya_%,xp_%,yp_%
    xa_%=xp_%,ya_%=yp_%
  NEXT i_%
RETURN
```

If you now click on the Maximizer the window extends to the full screen size. The Lissajous figure will, of course, not be redrawn. This will only happen if you insert the call to the **PROCEDURE** l_figur() after the **FULLW #1**:

Example:

```
          //
          DO
            GET_EVENT
            e%=MENU(1)
            SWITCH e%
            CASE 4                    // Closer event
              CLOSEW #1
            CASE 6                    // Maximizer event
              FULLW #1
***           l_figur()
            CASE 17                   // Mover event
              MOVEW #1,MENU(7),MENU(8)
            ENDSWITCH
          LOOP
          //
```

SIZEW changes the size of the window. The command requires three parameters: the window number, the new width and the new height of the window. The event which requires the usage of this command is clicking on the Sizer and changing of the window size while holding the mouse button down. This event sets **MENU(1)** to the value 18 and returns the new window width in **MENU(7)** and the new window height in **MENU(8)**. If after "sizing" the new window area is greater in either the X or Y direction, the fragments of the old window contents remain in the new area. Conse-

quently, the old fragments must be deleted and new window contents redrawn again. The GFA-BASIC command **PBOX** is used for this purpose. However, the fill colour and fill pattern must be defined first.

Example:

```
    SCREEN 16                       // EGA mode
    TITLEW#1," Window 1"
    INFOW #1,"Lissajous figure"
    OPENW#1,10,10,300,200,-1
    l_figur()
    DO
      GET_EVENT
      e%=MENU(1)
      SWITCH e%
      CASE 4                        // Closer event
        CLOSEW #1
      CASE 6                        // Maximizer event
        FULLW #1
        l_figur()
      CASE 17                       // Mover event
        MOVEW #1,MENU(7),MENU(8)
***   CASE 18                       // Sizer event
***     SIZEW #1,MENU(7),MENU(8)
***     l_figur()
      ENDSWITCH
    LOOP
    EDIT
    //
    PROCEDURE l_figur()
      LOCAL i_%,x_%,y_%,w_%,h_%
      LOCAL p_%,q_%,r_%
      LOCAL xa_%,ya_%,xp_%,yp_%
***   COLOR 0,0
***   DEFFILL 8
***   PBOX 0,0,_X,_Y
***   COLOR 15
      p_%=77,q_%=2,r_%=3
      x_%=0,y_%=0
```

```
    w_%=_X,h_%=_Y
    w_%>>=1,h_%>>=1
    xa_%=ADD(ADD(w_%,w_%),x_%)
    ya_%=ADD(h_%,y_%)
    FOR i_%=p_% TO 180*p_% STEP p_%
      xp_%=ADD(ADD(w_%,w_%*COSQ(q_%*i_%)),x_%)
      yp_%=ADD(ADD(h_%,h_%*SINQ(r_%*i_%)),y_%)
      LINE xa_%,ya_%,xp_%,yp_%
      xa_%=xp_%,ya_%=yp_%
    NEXT i_%
  RETURN
```

As long as you only use one window you can proceed in this way. But when the programs use several windows simultaneously, it frequently happens that when sizing a window the contents of another window become visible. They must also be redrawn. It's recommended therefore, after a **SIZEW**, to monitor for a redraw message (**MENU(1)=21** and redraw the changed window as well as the now visible parts of other windows (see below).

You may have been wondering how come that the Lissajou figure always corresponds to the window position and size. There are two reasons for this:

(1) When a window is opened, all graphic output (including the determination of the mouse position with **MOUSEX** and **MOUSEY** !!) are clipped to the window area. As long as this clipping is not changed with the **CLIP** command, the graphic cannot extend beyond the window area.

(2) The width of the inside area for the currently active window for graphic output is always in the GFA-BASIC variable **_X**, and the height of this area in **_Y**. The origins for all graphic output x=0, y=0 are always in the upper left corner of this window. If you check the variable initialisation in the **PROCEDURE** l_figur(), you'll notice that the variables p_%,q_% and r_% are initialised with

certain values. These values affect only the way the figure appears (try p_%=77,q_%=1,r_%=2). The origins for the graphic output are then set with x_%=0,y_%=0 to the upper left corner of the inner window area. The maximum area for the graphic output is then defined with w_%=**_X** and h_%=**_Y** to correspond to the horizontal and vertical size of the inner window area.

In this way, the graphics always fill the window and are also fully contained within it. If these parameters are changed, for example x_%=**_X**/4,y_%=**_Y**/4, the position and/or the size of the graphic in the window is also changed. This is needed when for example the window contents must be moved vertically or horizontally after a slider event.

To round off the discussion of all window elements we still need to address the Minimizer and the Sliders.

As the reaction to a click on the Minimizer the window should be shrunk to its minimum size, ideally so that it becomes an icon representative of the window. The window configuration parameters can be used for this purpose. If they are set to 0, only the inner area of the window is drawn with a 3D effect. To "iconise" a window in this manner it must first be closed and then again redrawn in a smaller size without any elements. If an icon is used, clicking on it should redraw the original window and delete the icon.

In the example program we'll insert the **PROCEDURE** iconize(), to create an icon when a window is closed and set the relevant flag. To inquire about a mouse click on the icon we monitor, from a **SWITCH...CASE** conditional statement, for the event **MENU(1)=3** which signals a mouse click on the active window. When this happens the window should again be reinitialised to its original form. Because of this, the initialisation procedure of the window (**TILEW...**, **INFOW...** and **OPENW...**) is moved to the **PROCEDURE** initialize_window().

Finally, the program should be able to terminate without Control + Cancel. This will occur when the window is closed. The **DO...LOOP** loop is therefore terminated when **MENU(1)** returns the value 4.

Example:

```
      SCREEN 16                        // EGA mode
***   xres%=_X,yres%=_Y
***   initialize_window()
      DO
        GET_EVENT
        e%=MENU(1)
        SWITCH e%
***     CASE 3                         // mouse click on active
      window
          IF icon!
            CLOSEW #1
            initialize_window()
            icon!=FALSE
          ENDIF
        CASE 4                         // Closer event
          CLOSEW#1
***     CASE 5                         // Minimizer event
          minimize()()
        CASE 6                         // Maximizer event
          FULLW #1
          l_figur()
        CASE 17                        // Mover event
          MOVEW #1,MENU(7),MENU(8)
        CASE 18                        // Sizer event
          SIZEW #1,MENU(7),MENU(8)
          l_figur()
        ENDSWITCH
***   LOOP UNTIL e%=4
      EDIT
      //
***   PROCEDURE initialize_window()
        TITLEW #1," Window 1"
        INFOW #1,"Lissajous figure"
```

```
      OPENW#1,10,10,300,200,-1
      l_figur()
    RETURN
    //
*** PROCEDURE minimize()
      CLOSEW #1
      OPENW #1,10,10,xres%/15,yres%/15,0
      l_figur()
      icon!=TRUE
    RETURN
    //
    PROCEDURE l_figur()
      LOCAL i_%,x_%,y_%,w_%,h_%
      LOCAL p_%,q_%,r_%
      LOCAL xa_%,ya_%,xp_%,yp_%
      CLEARW #1
      COLOR 0,0
      DEFFILL 8
      PBOX 0,0,_X,_Y
      COLOR 15
      p_%=77,q_%=1,r_%=2
      x_%=0,y_%=0
      w_%=_X,h_%=_Y
      w_%>>=1,h_%>>=1
      xa_%=ADD(ADD(w_%,w_%),x_%)
      ya_%=ADD(h_%,y_%)
      FOR i_%=p_% TO 180*p_% STEP p_%
        xp_%=ADD(ADD(w_%,w_%*COSQ(q_%*i_%)),x_%)
        yp_%=ADD(ADD(h_%,h_%*SINQ(r_%*i_%)),y_%)
        LINE xa_%,ya_%,xp_%,yp_%
        xa_%=xp_%,ya_%=yp_%
      NEXT i_%
    RETURN
```

Using these program modifications you can create an icon and then reproduce the original window again. Do note once more that shrinking (Minimize) or redrawing the window may reveal the covered parts of other windows and can, for example, report a Redraw event **(MENU(1) =21)**.

To inquire about the vertical and horizontal movements of window contents the following events are required

MENU(1) =7
MENU(1) =8
MENU(1) =11
MENU(1) =12
MENU(1) =15

MENU(1) =9
MENU(1) =10
MENU(1) =10
MENU(1) =14
MENU(1) =16

The first five events relate to the vertical and the second group of five to the horizontal movements. The idea behind this is that the slider movements always relate to the window. If a window area is moved up, the window remains graphically at the same coordinates. However, the contents of the window are moved down!

When using Sliders they should always move in the same direction as the windows. The position of the area displayed in the window in relation to the complete contents is shown as the position of the Slider within its box. The size of the Slider should also correspond to the size of the box.

Let's assume you have a window whose inner area is not big enough to encompass the picture which is to be drawn. You want, therefore, to be able to move the picture out of the window all the way up, down, left or right.

This means that you wish to use three _Y units in the vertical direction and three _X units in the horizontal direction. Since the whole slider area for the slider is in the range from 0 to 1000, at the start of the program the slider will have the size of 1000/3 = 333. The size and the position of the slider is set with the GFA-BASIC command **WINDSET** (see below).

MENU(1)=7 means that the field with the triangle pointing up was clicked on. The window must therefore be moved up, i.e the window contents must be moved down while the slider "walks" up.

MENU(1)=8 means that the field with the triangle pointing down was clicked on. The window must therefore be moved down, i.e the window contents must be moved up while the slider "walks" down.

A vertical step is thereby defined which is used as a unit for the movement. In the above example this step is 1/100 of the vertical dimension of the inner window area. The movement of the inner window area can the be represented with the slider. Regardless of the window size the slider area is always in the range from 0 to 1000. If the movement unit is set relative to this, the value 10 corresponds to a step of 1/100. The relevant variable (ve%) is defined for this purpose in the new **PROCEDURE** initialize_slider().

MENU(1)=11 means that the area above the vertical slider was clicked on. This causes a Page up to be performed. The window and the slider must thereby be moved up, while the window contents "walk" down.

MENU(1)=12 means that the area below the vertical slider was clicked on. This causes a Page down to be performed. The window and the slider must thereby be moved down, while the window contents "walk" up.

To simulate a Page up or Page down the previously set step is used to multiply with. In the example program below a factor of 10 is used and saved in the pv% variable for subsequent calculations.

MENU(1)=9 means that the field with the triangle pointing left was clicked on. The window must therefore be moved to the right, i.e the window contents must be moved left while the slider "walks" to the right.

MENU(1)=10 means that the field with the triangle pointing right was clicked on. The window must therefore be moved to the left, i.e the window contents must be moved right while the slider "walks" to the left.

A horizontal step is thereby defined which is used as a unit for the movement. In the above example this step is 1/100 of the horizontal dimension of the inner window area. The relevant variable (he%) is defined for this purpose in the new **PROCEDURE** initialize_slider().

MENU(1)=13 means that the area to the left of the horizontal slider was clicked on. This causes a Page left to be performed. The window and the slider must thereby be moved left, while the window contents are moved to the right.

MENU(1)=14 means that the area to the right of the horizontal slider was clicked on. This causes a Page right to be performed. The window and the slider must thereby be moved right, while the window contents are moved to the left.

To simulate a Page left or Page right the previously set step is used to multiply with. In the example program below a factor of 10 is used and saved in the ph% variable for subsequent calculations.

For the actual slider movement one monitors the events **MENU(1)=15** and **MENU(1)=16**. The area in which the slider moves, is in the range from 0 to 1000 and its value is returned after the relevant event in **MENU(7)**.

After a move the Slider should show the position of the window in the picture. To do this the slider must be set to the corresponding value. This is achieved with the GFA-BASIC command

WINDSET

where **WINDSET** 8,x redraws the vertical and **WINDSET** 10,x the horizontal position of the Sliders. The size of the slider at the beginning is also set with **WINDSET**. The call to set both sliders to the middle, when the sliders are 1/3 of the total area, is as follows:

```
WINDSET 8,500,333,500,333
```

Furthermore, the character **PROCEDURE** takes two Call by Value parameters which specify the point of origin for vertical and horizontal movements. This modifies all calls to this **PROCEDURE** and in the **PROCEDURE** itself the values are assigned to the variables x_% and y_%. x_% and y_% must be negated during the calculation of the slider sizes since they can be moved in both directions.

For clarity, the individual programming steps have been moved to the relevant **PROCEDURES** inside the **SWITCH...CASE** conditional statement.

Example:

```
SCREEN 16                      // EGA mode
xres%=_X,yres%=_Y              // resolution
initialize_window()            // get window parameters
initialize_slider()            // get scroll parameters
DO
  GET_EVENT                    // monitor event
  e%=MENU(1)
  SWITCH e%
  CASE 3                       // mouse click in active
  //                              window
    undo_icon()
  CASE 4                       // Closer event
    CLOSEW#1
```

```
  CASE 5                    // Minimizer event
    minimize()
  CASE 6                    // Maximizer event
    maximize()
  CASE 7,8                  // arrow up, arrow down
    slider_event()
  CASE 9,10                 // arrow left, arrow right
    slider_event()
  CASE 11,12                // Page up, Page down
    slider_event()
  CASE 13,14                // Page left, Page right
    slider_event()
  CASE 15,16                // vert-slider, hor-slider
    slider_event()
  CASE 17                   // Mover event
    MOVEW #1,MENU(7),MENU(8)
  CASE 18                   // Sizer event
    SIZEW #1,MENU(7),MENU(8)
    l_figur(fhpos%,fvpos%)
  ENDSWITCH
  WHILE MOUSEK
  WEND
LOOP UNTIL e%=4
EDIT
//
PROCEDURE initialize_window()
  TITLEW #1," Window 1 "
  INFOW #1,"Lissajous figure"
  OPENW#1,10,10,300,200,-1
  WINDSET 8,500,333,500,333
  l_figur(0,0)
RETURN
//
PROCEDURE initialize_slider()
  vpos%=500,hpos%=500
  ve%=10,he%=10
  pv%=100,ph%=100
```

```
RETURN
//
PROCEDURE undo_icon()
  IF icon!
    CLOSEW #1
    OPENW #1,xa%,ya%,wa%,ha%,-1
    l_figur(fhpos%,fvpos%)
    WINDSET 8,vsa%
    WINDSET 10,hsa%
    icon!=FALSE
  ENDIF
RETURN
//
PROCEDURE maximize()
  IF full!
    CLOSEW #1
    OPENW #1,xa%,ya%,wa%,ha%,-1
    l_figur(fhpos%,fvpos%)
    WINDSET 8,vsa%
    WINDSET 10,hsa%
  ELSE
    WINDGET 0,xa%,ya%,wa%,ha%
    WINDGET 8,vsa%
    WINDGET 10,hsa%
    FULLW #1
    l_figur(fhpos%,fvpos%)
    full!=TRUE
  ENDIF
RETURN
//
PROCEDURE minimize()
  WINDGET 0,xa%,ya%,wa%,ha%
  WINDGET 8,vsa%
  WINDGET 10,hsa%
  CLOSEW #1
  OPENW #1,10,10,xres%/15,yres%/15,0
  l_figur(0,0)
```

```
    icon!=TRUE
  RETURN
  //
  PROCEDURE slider_event()
    LOCAL vgo!,hgo!
    SWITCH e%
    CASE 7,11
      IF vpos%>0
        IF e%=7
          vpos%=MAX(vpos%-ve%,0)
        ELSE
          vpos%=MAX(vpos%-pv%,0)
        ENDIF
          vgo!=TRUE
      ENDIF
    CASE 8,12
      IF vpos%<1000
        IF e%=8
          vpos%=MIN(vpos%+ve%,1000)
        ELSE
          vpos%=MIN(vpos%+pv%,1000)
        ENDIF
        vgo!=TRUE
      ENDIF
    CASE 9,13
      IF hpos%>0
        IF e%=9
          hpos%=MAX(hpos%-he%,0)
        ELSE
          hpos%=MAX(hpos%-ph%,0)
        ENDIF
        hgo!=TRUE
      ENDIF
    CASE 10,14
      IF hpos%<1000
        IF e%=10
          hpos%=MIN(hpos%+he%,1000)
```

```
        ELSE
          hpos%=MIN(hpos%+ph%,1000)
        ENDIF
      hgo!=TRUE
      ENDIF
    CASE 15
      vpos%=MENU(7)
      vgo!=TRUE
    CASE 16
      hpos%=MENU(7)
      hgo!=TRUE
    ENDSWITCH
    //
    IF vgo!
      fvpos%=SCALE(vpos%,_Y,500)-_Y
      l_figur(fhpos%,fvpos%)
      WINDSET 8,vpos%
    ENDIF
    //
    IF hgo!
      fhpos%=SCALE(hpos%,_X,500)-_X
      l_figur(fhpos%,fvpos%)
      WINDSET 10,hpos%
    ENDIF
    //
  RETURN
  //
  PROCEDURE l_figur(xs_v%,ys_v%)
    LOCAL i_%,x_%,y_%,w_%,h_%
    LOCAL p_%,q_%,r_%
    LOCAL xa_%,ya_%,xp_%,yp_%
    COLOR 0,0
    PBOX 0,0,_X,_Y
    COLOR 15
    p_%=77,q_%=1,r_%=2
    x_%=-xs_v%,y_%=-ys_v%
    w_%=_X,h_%=_Y
```

```
    w_%>>=1,h_%>>=1
    a_%=ADD(ADD(w_%,w_%),x_%),ya_%=ADD(h_%,y_%)
    FOR i_%=p_% TO 180*p_% STEP p_%
      xp_%=ADD(ADD(w_%,w_%*COSQ(q_%*i_%)),x_%)
      yp_%=ADD(ADD(h_%,h_%*SINQ(r_%*i_%)),y_%)
      LINE xa_%,ya_%,xp_%,yp_%
      xa_%=xp_%,ya_%=yp_%
    NEXT i_%
  RETURN
```

After discussing the window elements we will now concentrate on the configuration parameters . As previously stated the configuration parameter is a word variable, which is used on a bit level. The set bit means the following:

0	vertical slider
1	triangles pointing up and down
2	horizontal slider
3	triangles pointing left and right
4	title bar
5	close box
6	minimum box
7	maximum box
8	info line
9	size box

If -1 is given as the configuration parameter all elements are drawn. If, in contrast, a 0 is given only a rectangle with a 3D effect is drawn. By deleting individual bits various window elements can be eliminated.

OPENW #1,10,10,300,200,~64 creates for example a window without the Minimizer.

OPENW #1,10,10,300,200,~256 creates a window without the Info line.

To create the elements in a window the GFA-BASIC command **OPENW** must always be used.

The individual elements, if already created, can be changed with the command

WINDSET

This includes - as shown in the above example - the setting of the position and size of the vertical and/or horizontal Sliders. The command **WINDSET** requires two parameters, **i** and **x**. The first parameter, i, indicates the window element to modify. The following applies:

i = 8	the position of the vertical slider in the range from 0 to 1000
i = 9	the size of the vertical slider in the range 0 to 1000
i = 10	the position of the horizontal slider in the range from 0 to 1000
i = 11	the size of the horizontal slider in the range from 0 to 1000
i = 13	sets the attributes for pushing of window buttons
i = 14	sets the character size for window writing (for example 8, 14, 16)
i = 15	the address of the character set.

The second parameter, x, specifies the actual value to use for modification.

```
OPENW #1,10,10,300,200,-1
WINDSET 8,500
```

sets, for example, the vertical slider in a window to the middle of the vertical slider area.

There is another important GFA-BASIC command which relates to the window parameters, the assignment

WINDGET

WINDGET serves to read the previously set window parameters. The command needs a minimum of two parameters, **i** and **x**. The first parameter, i, specifies the position in the parameter list where **WINDGET** should read from. x reads then a value from this position. The **WINDGET** parameter list is as follows:

i = 0	outside X coordinate of the window
i = 1	outside Y coordinates of the window
i = 2	outside width of the window
i = 3	outside height of the window
i = 4	X coordinate of the inner window area
i = 5	Y coordinate of the inner window area
i = 6	width of the inner window area = **_X**
i = 7	height of the inner window area = **_Y**
i = 8	position of the vertical slider in range from 0 to 1000
i = 9	size of the vertical slider in the range from 0 to 1000
i = 10	position of the horizontal slider in the range from 0 to 1000
i = 11	size of the vertical slider in the range from 0 to 1000
i = 12	reads the configuration word (previously set with **OPENW**)
i = 13	reading of the attribute of pressed window button (previously set with **WINDSET**)
i = 14	character set size (for example 8,14,16)
i = 15	the address of the character set
i = 16	the number of the top window (**TOPW**)
i = 17	the number of the second top window
i = 18	the number of the third top window
i = 19	the number of the bottom window.

The command **WINDGET** 16,x writes the number of the top window in variable x.

WINDGET 0,x,y,w,h writes the outside X coordinate of the window to x, the outside Y coordinate to y, the outside width to w and the outside height of the window to h.

To determine the number of the window at a certain coordinate intersection GFA-BASIC function

WINDFIND

should be used. This function can be invoked in two ways:

WINDFIND x,y,nr% and
nr%=WIND_FIND(x,y)

In both cases, after the call, the number of the window at the position specified with x and y is returned in the variable nr%. nr% can thereby have the values 0,1,2,3 and 4, where 0 indicates the Desktop. If there are several windows at the specified coordinates, nr% returns the number of the top window at this point.

Example:

```
SCREEN 18
OPENW 1,10,10,300,200,-1
OPENW 2,50,50,300,200,-1
WINDFIND 15,15,a%
WINDFIND 60,60,b%
WINDFIND 340,240,c%
SCREEN 3
COLOR 1
PRINT a%'b%'c%
REPEAT
UNTIL LEN(INKEY$)
```

The above example displays the values 1 2 2. The only window at position 15,15 is #1 and the only window at position 340,240 is window #2. The position 60,60 is above both window #1 as well as window #2. Since **OPENW** #2 was performed after **OPENW** #1 the second window is the Top Window, and **WINDFIND** returns in this case the value 2.

3.3.1 Redraw events

Whenever you open a window, close, activate, move it or change its size the used GFA-BASIC command (**OPENW, CLOSEW, TOPW, MOVEW, SIZEW** and **FULLW**) sends out an event message. It indicate which part of the Desktop must be restored. In the long example program above which monitors the sliders you can get the importance of such Redraw events when, for example, you wish to move a picture off screen and then bring it back. You will remember that the segment of the picture which is off screen cannot be restored with **MOVEW**.

When a Redraw message is sent, the event array **MENU()** contains the value **MENU(1)=21. MENU(7)** and **MENU(8)** contain then the X and Y coordinates, **MENU(9)** the width and **MENU(10)** the height of the rectangle which caused the message. To react to a Redraw message means to find out if there are overlaps between this "Redraw" rectangle and other rectangles on the Desktop.

The following commands are available in GFA-BASIC for this purpose:

GETFIRST
GETNEXT
RC_INTERSECT()

The GFA-BASIC command **GETFIRST** builds a rectangle list of overlapping rectangles in case of a Redraw event. The structure of this rectangle list is relatively complicated. The related area for this rectangle list is

always returned in **MENU(7)**, **MENU(8)**, **MENU(9)** and **MENU(10)**. Within this area will - if necessary - even the Desktop, i.e. window #0, internally restored by GFA-BASIC. All other windows or other graphics must, however, be inspected to see if they overlap with this area. The procedure can be sketched out as follows:

(1) Build a loop for all opened windows.

(2) Within this loop redirect the input and output by using **WIN #i** (i = loop index) for the relevant windows.

(3) Use **WINDGET** 4,x,y to get the X and Y coordinates of the upper left corner of the inner area for the i-th window.

(4) By using **GETFIRST #1,x,y,w,h** determine the X and Y coordinates, as well as the width (W) and height (H) of the first visible rectangle in this window.

(5) In another loop, test if the rectangle returned by **GETFIRST** is indeed one, i.e. test if the returned width and height are different from 0.

(6) If they are test by using the GFA-BASIC function **RC_INTERSECT**, if and which window areas have again become visible. If new window areas have become visible, the rectangle returned by **GETFIRST** and the Redraw rectangle are overlapping.

(7) Clip now the last four parameters returned by **RC_INTERSECT**, by the rectangle area. As offset for this clipping use the coordinate point received in (6).

(8) The graphic must now be redrawn in the clipping area.

(9) When that is done, the GFA-BASIC command **GETNEXT** is used to get the next rectangle for the i-th window.

(10) Repeat now the steps from point (5) until the loop cycles through all windows.

This procedure necessitates, naturally, that - under certain circumstances - a graphic must be drawn quite often. A program can be optimised in that the graphic routines get the areas only when the graphic has to be drawn from scratch.

The following example program draws two quite fancy graphics and employs the above outlined procedure. The main segment of the program is the **PROCEDURE** redraw().

```
/*-------------------- general  variables ------------------*/
//
nw%=2                                      /* number of windows
SCREEN 18                                  /* VGA mode
xres%=_X,yres%=_Y                          /* resolution
dc%=11                                     /* Desktop colour
DIM ICON!(2)                               /* icon 'flag'
def_window()                               /* window TYPE
OPENW #0                                   /* open Desktop
SYSCOL 6,1,dc%                             /* Desktop colour
CLEARW #0                                  /* paint Desktop
//
/*-------------------- draw picture in window 1. ---------*/
//
DIM ww(50,50)                              /* original array
DIM w(50,50)                               /* its working copy
f.n=1                                      /* window 1.
window_param()                             /* window parameters
f1()                                       /* ww() initialise
TITLEW #f.n,f.t$                           /* window title
```

```
OPENW #f.n,f.x,f.y,f.w,f.h,~15             /* open a window without
                                              sliders
KILL_EVENT                                 /* delete event, draw there
window_set()                               /* parameters for drawing
ff1()                                      /* further parameters +
//                                            drawing
//
/*--------------------- draw picture in window 2. --------*/
//
aw%=4                                      /* number of values per month
am%=12                                     /* number of months
min%=10                                    /* minimum value
max%=110                                   /* maximum value
DIM values%(am%,aw%)                       /* original array
f.n=2                                      /* window 2.
window_param()                             /* window parameters
g1()                                       /* initialise values%()
TITLEW #f.n,f.t$                           /* window title
OPENW #f.n,f.x,f.y,f.w,f.h,~15             /* open window without
//                                            sliders
KILL_EVENT                                 /* delete event, draw there
window_set()                               /* parameters for drawing
pp2()                                      /* draw picture
//
/*-------------------- event inquiry ------------------------*/
//
DO                                         /* event inquiry
  GETEVENT
  e%=MENU(1)
  EXIT IF e%=4                             /* activate close box
  IF e%=21
    redraw()                               /* call Redraw routine
  ELSE
    evaluate_message()                     /* call evaluation routine
  ENDIF
```

```
LOOP
EDIT                                       /* back to the editor
//
/*--------------------- general routines -------------------*/
//
PROCEDURE def_window()                     /* Window TYPE
  TYPE window:                             /* its TYPE
  - CHAR*20 t$                             /* window title
  - CHAR*20 i$                             /* info title
  - UBYTE n                                /* window number
  - CARD x                                 /* X position
  - CARD y                                 /* Y position
  - CARD w                                 /* width
  - CARD h                                 /* height
  ENDTYPE                                  /* TYPE end
  window:f.                                /* the TYPE variable
RETURN
//
PROCEDURE window_param()                   /* Window parameters
  f.t$=" Window "+STR$(f.n)+ " "           /* window title
  f.w=xres%/2-15                           /* window width
  f.h=yres%/2-5                            /* window height
  f.y=60                                   /*  Y position
  SWITCH f.n
  CASE 1
    f.x=10                                 /* X position for window 1.
  CASE 2
    f.x=f.w+20                             /* X position for window 2.
  ENDSWITCH
RETURN
//
PROCEDURE window_set()                     /* inner area of the window
  LOCAL w_%,h_%
  f.x=0
  f.y=0
  WINDGET 6,w_%,h_%
  f.w=w_%
```

```
  f.h=h_%
RETURN
//
PROCEDURE evaluate_message()             /* event evaluation
  LOCAL ev_%
  SWITCH e%
  CASE 2                                 /* mouse click outside of Top
    //                                      window
    ev_%=MENU(7)                         /* determine window under
    //                                      mouse
    IF ev_%                              /* window under mouse ?
      f.n=ev_%                           /* window number
      IF ICON!(f.n)                      /* window is an icon
        CLOSEW #f.n                      /* close icon
        window_param()                   /* get window parameters
        TITLEW #f.n,f.t$                 /* title line
        OPENW #f.n,f.x,f.y,f.w,f.h,~15   /* open window
        window_set()                     /* get graphic coordinates
        ICON!(f.n)=FALSE                 /* set icon flag to FALSE
        KILL_EVENT                       /* delete event, draw new
        SWITCH f.n                       /* which window ?
        CASE 1                           /* #1
          ff1()                          /* draw picture 1
        CASE 2                           /* #2
          pp2()                          /* draw picture 2
        ENDSWITCH
      ELSE                               /* window is not an icon
        TOPW #f.n                        /* redirect output to window
        window_set()
      ENDIF
    ENDIF
  CASE 5                                 /* Minimizer event
    CLOSEW #f.n                          /* close window
    ICON!(f.n)=TRUE                      /* set icon flag
    SWITCH f.n
    CASE 1
      OPENW #1,10,10,40,40,0             /* open little window as icon
```

```
      f.n=2                              /* divert to second window
    CASE 2
      OPENW #2,60,10,40,40,0             /* open little window as icon
      f.n=1                              /* divert to first window
    ENDSWITCH
    window_set()                         /* get window parameters
    WIN #f.n
  CASE 6                                 /* Maximize event
    FULLW #f.n                           /* window to maximum size
    window_set()                         /* get window parameters
  CASE 17                                /* Mover event
    MOVEW #f.n,MENU(7),MENU(8)           /* move window
    window_set()                         /* get window parameters
  CASE 18                                /* Sizer event
    SIZEW #f.n,MENU(7),MENU(8)           /* change window size
    window_set()                         /* get window parameters
  ENDSWITCH
RETURN
//
PROCEDURE redraw()                       // Redraw routine
  LOCAL i_%,x_%,y_%,w_%,h_%
  LOCAL rx_%,ry_%,rw_%,rh_%
  x_%=MENU(7), y_%=MENU(8)               /* redraw rectangle
  w_%=MENU(9), h_%=MENU(10)
  FOR i_%=1 TO nw%                       /* window loop
    WIN #i_%                             /* divert to window #i_%
    GETFIRST #i_%,rx_%,ry_%,rw_%,rh_%    /* start of rectangle list
    WHILE rw_% && rh_%                   /* as long as there is a rec-
      //                                    tangle
      IF RC_INTERSECT(x_%,y_%,w_%,h_%,rx_%,ry_%,rw_%,rh_%)
        WINDGET 4,xx%,yy%                /* upper left coordinate
        window_set()                     /* get window parameters
        CLIP rx_%,ry_%,rw_%,rh_% OFFSET xx%,yy%    /*  overlapping area
        SWITCH i_%                       /* which window is the redraw
        CASE 1                           /* for window 1
          ff1()                          /* draw picture 1 again
        CASE 2                           /* window 2
```

```
          pp2()                              /* draw picture 2 again
        ENDSWITCH
      ENDIF
      GETNEXT rx_%,ry_%,rw_%,rh_%            /* get other rectangles
    WEND
  NEXT i_%                                   /* next window
  WIN #f.n                                   /* divert to TOPW
RETURN
//
/*----------------- picture for the first window --------------------*/
//
PROCEDURE f1()                               /* coordinates for SIN-COS
  //                                            lattice
  LOCAL i%,j%
  FOR i%=0 TO 360 STEP 18
    FOR j%=0 TO 360 STEP 18
      ww(i%/18+1,j%/18+1)=COSQ(i%)-SINQ(j%)
    NEXT j%
  NEXT i%
RETURN
//
PROCEDURE ff1()                              /* set working copy and draw
  MAT CPY w()=ww()
  pp1(21,21,-2,2)
RETURN
//
PROCEDURE pp1(z_v%,s_v%,za_v,ze_v)           /* picture 1.
  LOCAL i_%,j_%,k_%,z_%
  LOCAL xw_%,xg_%,yg_%,zg_%,xp_%,yp_%
  LOCAL ppz_,pps_,ppe_,sx_,sy_,cx_,cy_
  LOCAL vx_,vy_,vz_,wx_,wy_,i1p_,j1p_
  //
  COLOR 4,4                                  /* red background
  DEFFILL 8
  PBOX 0,0,9999,9999
  //
  wx_=35,wy_=35                              /* offset variables
```

```
xw_%=f.w/2
z_%=f.h*(25/40)
xg_%=z_%,yg_%=z_%,zg_%=z_%
xp_%=ADD(f.x,xw_%)
yp_%=ADD(f.y,f.h*(29/40))
ppz_=xg_%/z_v%,pps_=zg_%/s_v%
ppe_=yg_%/(ze_v-za_v)
//
FOR i_%=0 TO z_v%
  w(i_%,0)=za_v
NEXT i_%
//
FOR j_%=1 TO s_v%
  w(0,j_%)=za_v
NEXT j_%
//
FOR i_%=0 TO z_v%
  FOR j_%=0 TO s_v%
    w(i_%,j_%)=MAX(w(i_%,j_%),za_v)
    SUB w(i_%,j_%),za_v
  NEXT j_%
NEXT i_%
//
DIM px(4),py(4),pz(4)
sy_=SINQ(wy_),cy_=COSQ(wy_)
sx_=SINQ(wx_),cx_=COSQ(wx_)
DIV xg_%,2
DIV yg_%,2
DIV zg_%,2
//
FOR i_%=PRED(z_v%) DOWNTO 0
  i1p_=i_%*ppz_+ppz_
  FOR j_%=PRED(s_v%) DOWNTO 0
    j1p_=j_%*pps_+pps_
    IF i_%=0
      px(0)=i1p_
    ELSE
```

```
    px(0)=i_%*ppz_
  ENDIF
  py(0)=yg_%-w(i_%,j_%)*ppe_
  IF j_%=0
    pz(0)=j1p_
  ELSE
    pz(0)=j_%*pps_
  ENDIF
  px(1)=i1p_
  py(1)=yg_%-w(i_%+1,j_%)*ppe_
  IF j_%=0
    pz(1)=j1p_
  ELSE
    pz(1)=j_%*pps_
  ENDIF
  px(2)=i1p_
  py(2)=yg_%-w(i_%+1,j_%+1)*ppe_
  pz(2)=j1p_
  IF i_%=0
    px(3)=i1p_
  ELSE
    px(3)=i_%*ppz_
  ENDIF
  py(3)=yg_%-w(i_%,j_%+1)*ppe_
  pz(3)=j1p_
  //
  FOR k_%=0 TO 3
    vx_=px(k_%)-xg_%,vz_=pz(k_%)-zg_%
    px(k_%)=cy_*vx_-sy_*vz_+xp_%
    pz(k_%)=sy_*vx_+cy_*vz_+zg_%
  NEXT k_%
  FOR k_%=0 TO 3
    vy_=py(k_%)-yg_%,vz_=pz(k_%)-zg_%
    py(k_%)=cx_*vy_-sx_*vz_+yp_%
  NEXT k_%
  COLOR 14
  POLYFILL 5,px(),py()
```

```
      COLOR 1
      POLYLINE 4,px(),py()
    NEXT j_%
  NEXT i_%
  ERASE px(),py(),pz()
RETURN
//
/*-------------- picture for the second window --------------------*/
//
PROCEDURE g1()                            /* set random values
  LOCAL i_%,j_%
  FOR i_%=1 TO am%
    FOR j_%=1 TO aw%
      values%(i_%,j_%)=RAND(max%+min%)
    NEXT j_%
  NEXT i_%
RETURN
//
PROCEDURE pp2()                           /* picture 2.
  LOCAL i_%,j_%,bs_%
  LOCAL i_,w_,xs_,xe_,ys_,ye_,yy_,zs_,ze_
  //
  COLOR 15,15                             /* white background
  DEFFILL 8
  PBOX 0,0,9999,9999
  //
  DIM bdx%(4),bdy%(4),bsx%(4),bsy%(4)
  //
  bs_%=(f.w-8)/30                         /* width
  yy_=f.h/2+f.h/20                        /* lattice height
  w_=55                                   /* view angle
  COLOR 4                                 /* lattice colour
  /* f.x                                  /* X axis start
  /* f.y                                  /* Y axis start
  /* am%                                  /* X axis division
  zs_=1                                   /* Z axis start
  ze_=8                                   /* Z axis end
```

```
//
/*-------- draw horizontal segment ----------*/
//
FOR i_%=yy_ TO 0 STEP -20
  //
  xkord(f.x,ze_,w_,xs_)                 /* back
  ykord(i_%,ze_,w_,ys_)
  xkord(am%-0.5,ze_,w_,xe_)
  //
  LINE xs_,ys_,xe_,ys_
  //
  xkord(f.x,zs_,w_,xs_)                 /* side
  xkord(f.x,ze_,w_,xe_)
  ykord(i_%,zs_,w_,ys_)
  ykord(i_%,ze_,w_,ye_)
  //
  LINE xs_,ys_,xe_,ye_
  //
NEXT i_%
//
/*-------- draw vertical segment ------------*/
//
FOR i_=0.5 TO am%-0.5
  //
  xkord(i_,zs_,w_,xs_)                  /* back
  xkord(i_,ze_,w_,xe_)
  ykord(f.y,zs_,w_,ys_)
  ykord(f.y,ze_,w_,ye_)
  //
  LINE xs_,ys_,xe_,ye_
  //
  xkord(i_,ze_,w_,xs_)                  /* bottom
  ykord(f.y,ze_,w_,ys_)
  ykord(yy_,ze_,w_,ye_)
  //
  LINE xs_,ys_,xs_,ye_
  //
```

```
NEXT i_
//
/*--------- draw horizontal segment ---------*/
//
FOR i_%=ze_ DOWNTO zs_
  //
  xkord(f.x,i_%,w_,xs_)                /* bottom
  ykord(f.y,i_%,w_,ys_)
  xkord(am%-0.5,i_%,w_,xe_)
  //
  LINE xs_,ys_,xe_,ys_
  //
  xkord(f.x,i_%,w_,xs_)                /* and vertical segment
  xe_=xs_                              /* side
  ykord(f.y,i_%,w_,ys_)
  ykord(yy_,i_%,w_,ye_)
  //
  LINE xs_,ys_,xs_,ye_
  //
NEXT i_%
//
/*---------- draw beams --------------------*/
//
FOR i_%=0 TO am%-1                     /* for am% months
  FOR j_%=aw% DOWNTO 1                 /* with aw% values per month
    //
    //                                 /* front view
    //
    xkord(i_%,j_%,w_,xs_)              /* upper left corner of the
    //                                    beam
    ykord(f.y,j_%,w_,ys_)
    xkord(i_%,j_%,w_,xe_)              /* lower right corner of the
    //                                    beam
    //
    xe_+=bs_%
    y_=values%(i_%+1,aw%-j_%+1)*(f.h/2/max%)
    ykord(y_,j_%,w_,ye_)
```

```
      //
      // -------------- beam cover ---------------
      //
      bdx%(0)=SUCC(xs_),bdy%(0)=PRED(ye_)
      bdx%(1)=xs_+9,bdy%(1)=ye_-9
      bdx%(2)=xs_+9+bs_%-2,bdy%(2)=ye_-9
      bdx%(3)=xs_+bs_%-1,bdy%(3)=PRED(ye_)
      bdx%(4)=SUCC(xs_),bdy%(4)=PRED(ye_)
      //
      COLOR 14                          /* colour of the beam cover
      DEFFILL 8                         /* beam cover fill pattern
      POLYFILL 5,bdx%(),bdy%()          /* draw beam cover
      //
      COLOR 1                           /* beam front colour
      DEFFILL 8                         /* fill pattern of beam front
      PBOX xs_+1,ys_+1,xe_-1,ye_-1      /* draw beam front
      //
      // ------------- beam side -------------
      //
      bsx%(0)=xs_+9+bs_%-2,bsy%(0)=ye_-9
      bsx%(1)=xs_+9+bs_%-2,bsy%(1)=ys_-9
      bsx%(2)=xs_+PRED(bs_%),bsy%(2)=SUCC(ys_)
      bsx%(3)=xs_+PRED(bs_%),bsy%(3)=PRED(ye_)
      bsx%(4)=xs_+9+bs_%-2,bsy%(4)=ye_-9
      //
      COLOR 12                          /* beam side colour
      DEFFILL 8                         /* beam side fill pattern
      POLYFILL 5,bsx%(),bsy%()          /* draw beam pattern
      //
    NEXT j_%
    //
  NEXT i_%
  //
  ERASE bdx%(),bdy%(),bsx%(),bsy%()
  //
RETURN
//
```

```
PROCEDURE xkord(x,z,w, VAR tt)
  tt=((f.w-8)/14*x)+(f.h/18*z)*COSQ(w)
RETURN
//
PROCEDURE ykord(y,z,w, VAR tt)
  tt=(f.h-5)-y-(f.h/18*z)*SINQ(w)
RETURN
```

4. System routines

The PC operating system routines are invoked using interrupts. The 8086 processor and its successors offer 256 different interrupts numbered from $00 to $FF. In part, they are initiated by the hardware, such as the keyboard interrupt, and in part by the software, such as the **INTR($21)** = MS-DOS. Only a few interrupts are useful for programs, since most are intended for access of exotic hardware.

The interrupts are invoked using hexadecimal numbers by convention, although any other integer expression can be used with **INTR()**.

The most important software interrupts are:

BIOS

INTR($10)	video processing, EGA cards etc.
INTR($11)	get configuration
INTR($12)	get memory size
INTR($13)	low level disk access
INTR($14)	serial I/O
INTR($15)	"cassette", and sundry
INTR($16)	keyboard
INTR($17)	parallel I/O

DOS

INTR($21)	MS-DOS = all DOS functions
INTR($25)	read disk sector(s)
INTR($26)	write disk sector(s)

Expansions

INTR($33)	mouse
INTR($67)	EMS (Expanded Memory System)

A few examples:

4.1 Keyboard

Read character

~INTR($16,_AH=0)

This interrupt reads a character from the keyboard. This call corresponds to **~INP(2)** in older ST versions of GFA-BASIC frequently used when waiting for a keypress. **KEYGET** dummy% should be used instead to maintain portability.

Extended character read

~INTR($16,_AH=16)

This interrupt reads a character from the keyboard. In contrast to **INTR($16,_AH=0)** this call not only returns the letter codes but also the codes for F11 and F12 as well as the alternate codes. More keys can therefore be interrogated on an AT:

Example:

```
A%=INTR($16,_AH=16)
SELECT a%
CASE $85
  PRINT "Function key F11 pressed"
CASE $86
  PRINT "Function key F12 pressed"
CASE $87
  PRINT "Shift function key F11 pressed"
CASE $88
  PRINT "Shift function key F12 pressed"
CASE $89
  PRINT "Control function key F11 pressed"
```

```
CASE $8a
  PRINT "Control function key F12 pressed"
CASE $8b
  PRINT "Alternate function key F11 pressed"
CASE $8c
  PRINT "Alternate function key F12 pressed"
ENDSELECT
```

Report shift status

a%=INTR($16,_AH=2)

This interrupt tests the keyboard shift flags. It can report if the Shift, Alternate etc. are pressed. This call corresponds to a%=**BIOS**(11,-1) in GFA-BASIC ST.

Status of special keys

~INTR($16,_AH=$02)

Example:

```
~INTR($16,_AH=$02)
IF BTST(_AL,0)
  PRINT "Right shift key pressed"
ENDIF
IF BTST(_AL,1)
  PRINT "Left shift key pressed"
ENDIF
IF BTST(_AL,2)
  PRINT "Ctrl key pressed"
ENDIF
IF BTST(_AL,3)
  PRINT "Alt key pressed"
ENDIF
IF BTST(_AL,4)
```

```
  PRINT "Scroll-Lock on"
ENDIF
IF BTST(_AL,5)
  PRINT "Num-Lock on"
ENDIF
IF BTST(_AL,6)
  PRINT "Caps-Lock on"
ENDIF
IF BTST(_AL,7)
  PRINT "Insert mode on"
ENDIF
```

Reports the status of special keys and returns relevant information.

4.2 Serial port

Initialisation

~INTR($14,_AH=0,_AL=par,_DX=n)

This call is used to set the communication parameters for one of the serial ports. **_DX** specifies the number of the port (0=COM1..3=COM4). **_AL** contains the parameters to be set.

Bit 0 - 1	= 10	7 bits
	= 01	8 bits
Bit 2	= 0	one stop bit
	= 1	depending on the baud rate 1.5 or 2 stop bits
Bit 3 - 4	= 00	no parity
	= 01	odd parity
	= 11	even parity
Bit 5 - 7	= 000	110 baud
	= 001	150 baud
	= 010	300 baud
	= 011	600 baud
	= 100	1200 baud
	= 101	2400 baud
	= 110	4800 baud
	= 111	9600 baud

After the function call **_AH** contains the status of the serial port and **_AL** the modem status. The set bits in **_AH** will then indicate the following:

Bit 0	data ready
Bit 1	data overflow
Bit 2	parity error
Bit 3	protocol not observed

Bit 4	break
Bit 5	transmission hold register empty
Bit 6	transmission shift register empty
Bit 7	timeout

The set bits in **_AL** indicate:

Bit 0	modem ready to send
Bit 1	modem on
Bit 2	ring
Bit 3	connection with receiving modem

The bits 0 to 3 are copied to bits 4 to 7. The relevant status can be interrogated with

```
~INTR($14,_AH=$03, _DX=n)
```

Example:

```
~INTR($14,_AH=0,_AL=%10100010,_DX=000)
```

Configures COM 1 with 7 bits, on stop bit, no parity check and 2400 baud.

Inquire status

~INTR($14,_AH=3,_DX=n)

Example:

```
~INTR($14,_AH=3,_DX=0)
```

tests the status of COM 1.

The set bits in **_AH** will then indicate the following:

Bit 0	data ready
Bit 1	data overflow
Bit 2	parity error

Bit 3 protocol not observed
Bit 4 break
Bit 5 transmission hold register empty
Bit 6 transmission shift register empty
Bit 7 timeout

The set bits in _AL indicate:

Bit 0 modem ready to send
Bit 1 modem on
Bit 2 ring
Bit 3 connection with receiving modem

The bits 0 to 3 are copied to bits 4 to 7.

Read characters

~INTR($14,_AH=2)

Example:

```
~INTR($14,_AH=2,_DX=0) // test status
IF BTST(_AH,0)         // character ready
  ~INTR($14,_AH=2)     // read character
  IF BTST(_AH,7)=0     // character received
    PRINT CHR$(_AL)
  ELSE                 // error message
    PRINT "character not received"
  ENDIF
ENDIF
```

Reads a character from the serial port, if available, and displays it on the screen.

This function should always be invoked first when

```
~INTR($14,_AH=3)
```

reports that a character is ready to be received. In this case bit 0 in **_AH** is set.

If after calling this function the bit 7 in **_AH** is clear, the sent character was received. The ASCII code of the character is then in **_AL**.

If after calling this function the bit 7 in **_AH** is set, an error has occurred. The bits 0 to 6 in **_AH** then contain the status of the serial port. The set bits then indicate the following:

Bit 0	data ready
Bit 1	data overwritten
Bit 2	parity error
Bit 3	protocol not observed
Bit 4	break
Bit 5	transmission hold register empty
Bit 6	transmission shift register empty

Character output

~INTR($14,_AH=1,_AL=m,_DX=n)

Example:

```
FOR i%=65 TO 90
  _AL=i%
  _DX=0
  ~INTR($14,_AH=1)
NEXT i%
```

Sends all capital letters from the US character set to the serial port.

If after calling this function the bit 7 in _AH = 0, the characters were transmitted successfully.

If after calling this function the bit 7 in _AH is set, an error has occurred. The bits 0 to 6 in _AH then contain the status of the serial port. The set bits then indicate the following:

Bit 0	data ready
Bit 1	data overwritten
Bit 2	parity error
Bit 3	protocol not observed
Bit 4	break
Bit 5	transmission hold register empty
Bit 6	transmission shift register empty

4.3 Printer (parallel) port

Printer initialisation

~INTR($17,_AH=1,_DX=n)

This function intialises a printer connected to one of the parallel ports. It should always be called before sending the first character.

After the function call **_AH** contains the printer status. The set bits then indicate the following:

Bit 0	timeout
Bit 3	transfer error
Bit 4	printer on line
Bit 5	no paper
Bit 6	reception not confirmed
Bit 7	printer not busy

Example:

```
~INTR($17,_AH=1,_DX=0)
IF BTST(_AH,0)
  PRINT "Timeout"
EDNIF
IF BTST(_AH,3)
  PRINT "Communication error"
ENDIF
IF BTST(_AH,4)=0
  PRINT "Printer not ON line"
ENDIF
IF BTST(_AH,5)
  PRINT "Insert paper, please"
ENDIF
IF BTST(_AH,6)=0
  PRINT "Reception not confirmed"
ENDIF
```

```
IF BTST(_AH,7)=0
  PRINT "Printer busy"
ENDIF
```

Initialises the printer, tests its status and displays it.

Inquire printer status

~INTR($17,_AH=2,_DX=n)

After the function call _AH contains the printer status. The set bits then indicate the following:

Bit 0	timeout
Bit 3	transfer error
Bit 4	printer on line
Bit 5	no paper
Bit 6	reception not confirmed
Bit 7	printer not busy

Example:

```
~INTR($17,_AH=2,_DX=0)
IF BTST(_AH,0)
  PRINT "Timeout"
EDNIF
IF BTST(_AH,3)
  PRINT "Communication error"
ENDIF
IF BTST(_AH,4)=0
  PRINT "Printer not ON line"
ENDIF
IF BTST(_AH,5)
  PRINT "Insert paper, please"
ENDIF
IF BTST(_AH,6)=0
  PRINT "Reception not confirmed"
```

```
ENDIF
IF BTST(_AH,7)=0
  PRINT "Printer busy"
ENDIF
```

Checks the printer status and, optionally, reports the error.

Output characters

~INTR($17,_AH=0,_AL=m,_DX=n)

Example:

```
~INTR($17,_AH=0,_DX=0)     // initialise printer
FOR i%=65 TO 90
  _AL=i%
  ~INTR($23,_AH=0,_DX=0)   // output characters
NEXT i%
```

Initialise printer on LPT 1 and display all capital characters from the US character set.

4.4 Joystick

Status of the fire button

~INTR($15,_AH=$84,_DX=$0)

After this function call, **BTST(_FL,0)** returns 0 if there is no game adapter with the accompanying joysticks, or 1 for no error. In this case (**BTST(_AL,0)=1**) reads from **_AL** the status of the fire button. The set bits then indicate the following:

Bit 4	second fire button on second joystick pressed
Bit 5	first fire button on second joystick pressed
Bit 6	second fire button on first joystick pressed
Bit 7	first fire button on first joystick pressed

Example:

```
DO
  a$=INKEY$
  ~INTR($15,_AH=$84,_DX=0)
  IF BTST(_FL,0)=0
    PRINT "Error"
  ELSE
    IF BTST(_AL,4)
      PRINT "Fire button 2 on joystick 2 down"
    ENDIF
    IF BTST(_AL,5)
      PRINT "Fire button 1 on joystick 2 down"
    ENDIF
    IF BTST(_AL,6)
      PRINT "Fire button 2 on joystick 1 down"
    ENDIF
    IF BTST(_AL,7)
```

```
    PRINT "Fire button 1 on joystick 1 down"
  ENDIF
 ENDIF
LOOP UNTIL a$=CHR$(27)
```

Tests the status of the fire buttons of the joystick(s) and prints either this status or the error message.

Joysticks position

~INTR($15,_AH=$84,_DX=1)

After this function call, **BTST(_FL,0)** returns 0 if there is no game adapter with the accompanying joysticks, or 1 for no error. In this case **(BTST(_AL,0)=1)** the following registers contain (in pixels):

_AX	X coordinate of first joystick
_BX	Y coordinate of first joystick
_CX	X coordinate of second joysticks
_DX	y coordinate of second joysticks

Example:

```
DO
  a$=INKEY$
  ~INTR($15,_AH=$84,_DX=1)
  PRINT AT(10,10);Space$(69)
  IF BTST(_FL,0)
    PRINT AT(10,10);_AX''_BX
    PRINT AT(10,60);_CX''_DX
  ENDIF
LOOP UNTIL a$=CHR$(27)
```

Assuming there are no errors, it displays the x and y coordinates of the first joysticks on the left, and the second joystick on the right side of the screen.

4.5 Read disk sectors

~ INTR($25,_AL = drv,_DS=swap(addr),_BX=addr,_CX=count, _DX=secnum)

This call reads from drive **_AL** (0 = A, 1 = B..) starting with sector **_DX**, **_CX** sectors into a buffer at address **_DS:_BX**.

This example searches a portion of the hard disk partition E for the string "TEST" and displays the numbers of the sectors where this string occurs.

Example:

```
drv%=ASC("E")-65
FOR i%= 1000 TO 1120
  secread(i%)
  IF INSTR(UPPER$(a$),"TEST")
    PRINT i%
  ENDIF
NEXT i%
PROCEDURE secread(i%)
  _AL = drv%
  a$=SPACE$(512)
  _BX = V:a$
  _DS = SWAP(V:a$)
  _CX = 1
  _DX = i%
  ~INTR($25)
RETURN
```

DOS Interrupts

Interrupt	Function	Description
$10	0	VIDEO - define video mode
	1	VIDEO - define cursor type
	2	VIDEO - set cursor position
	3	VIDEO - get cursor position
	4	VIDEO - returns lightpen position
	5	VIDEO - select screen page
	6	VIDEO - scroll screen up
	7	VIDEO - scroll screen down
	8	VIDEO - read character with attribute
	9	VIDEO - write character with attribute
	0A	VIDEO - write character at cursor position
	0B	VIDEO - set colour palette
	0C	VIDEO - set pixel
	0D	VIDEO - get pixel
	0E	VIDEO - write text in teletype mode
	0F	VIDEO - returns current video mode
	10	VIDEO - set palette register
	11	VIDEO - character generator
	12	VIDEO - function call
	13	VIDEO - write string
	14	VIDEO - load LCD character set
	15	VIDEO - return physical parameter
	16	RESERVED
	17	RESERVED
	18	RESERVED
	19	RESERVED
	1A	VIDEO - read/write hardware configuration
	1B	VIDEO - return video status
	1C	VIDEO - save/restore VIDEO status
	1D-FF	RESERVED
$11	-	return system configuration
$12	-	memory size

DOS Interrupts

Interrupt	Function	Description
$13	0	DISK DRIVE - system initialization
	1	DISK DRIVE - read status
	2	DISK DRIVE - read from disk
	3	DISK DRIVE - write to disk
	4	DISK DRIVE - test sectors
	5	DISK DRIVE - format track
	6	DRIVE - format cylinder and set bad sector flags
	7	DRIVE - format drive and start with cylinder
	8	DRIVE - return current drive parameters
	9	DRIVE - install drive
	0A	DRIVE - read extended sectors
	0B	DRIVE - write extended sectors
	0C	DRIVE - search for cylinder
	0D	DRIVE - additional disk reset
	0E	DRIVE - read sector buffer
	0F	DRIVE - write sector buffer
	10	DRIVE - test if drive is ready
	11	DRIVE - park heads
	12	DRIVE - RAM test controller
	13	DRIVE - drive test
	14	DRIVE - controller test
	15	DRIVE - read DASD type
	16	DISK DRIVE - return disk change status
	17	DISK DRIVE - set DASD type for formatting
	18	DISK DRIVE - set media type
	19	DRIVE - park heads
	1A	DRIVE - format unit
	1B-FF	RESERVED
$14	0	SERIAL - initialize communication port
	1	SERIAL - send character to serial port
	2	SERIAL - receive character from serial port
	3	SERIAL - port status

DOS Interrupts

Interrupt	Function	Description
$14	4	SERIAL - extended initialization
	5	SERIAL - extended port control
	6-FF	RESERVED
$15	0	CASSETTE - motor on
	1	CASSETTE - motor off
	2	CASSETTE - read data block
	3	CASSETTE - write data block
	4-0E	RESERVED
	0F	DRIVE - periodic formatting
	10-1F	RESERVED
	20	AL=10 SYSREQ on AL=11 SYSREQ off
	21	DEVICE - self test error list
	22-3F	RESERVED
	40	DEVICE - read/test (profile)
	41	DEVICE - wait for external event
	42	DEVICE - system power supply OFF (Request)
	43	DEVICE - read system status
	44	DEVICE - activate internal modem power supply
	45-4E	RESERVED
	4F	KEYBOARD - capture keyboard
	50-7F	RESERVED
	80	DEVICE - open device
	81	DEVICE - close device
	82	DEVICE - terminate program
	83	DEVICE - interval wait
	84	JOYSTICK
	85	SYSTEM - System Request key
	86	DEVICE - wait
	87	DEVICE - move block
	88	MEMORY - return size of extended memory
	89	MEMORY - switch to protected mode

DOS Interrupts

Interrupt	Function	Description
$15	90	DEVICE - device busy
	91	DEVICE - turn interrupt off and set flag
	92-BF	RESERVED
	C0	DEVICE - return system parameters
	C1	DEVICE - segment address of extended BIOS
	C2	DEVICE - BIOS interface for handle-DEVICE
	C3	DEVICE - activate timeout watch-dog
	C4	DEVICE - programable random selection
	C5-FF	RESERVED
$16	0	KEYBOARD - read characters from keyboard
	1	KEYBOARD - return keyboard status
	2	KEYBOARD - return keyboard flags
	3	KEYBOARD - delay
	4	KEYBOARD - key click ON/OFF
	5	KEYBOARD - write characters
	6-0F	RESERVED
	10	KEYBOARD - extended character read from keyboard
	11	KEYBOARD - extended key inquiry
	12	KEYBOARD - return extended shift status
	13-FF	RESERVED
$17	0	PRINTER - send character to printer
	1	PRINTER - initialize printer port
	2	PRINTER - set printer status
	3-FF	RESERVED
$18	-	BASIC - load Basic
$19	-	BOOTSTRAP - load boot sector
$1A	0	TIMER - read time
	1	TIMER - set time
	2	TIMER - get RTC clock
	3	TIMER - set RTC clock
	4	TIMER - get RTC date

DOS Interrupts

Interrupt	Function	Description
$1A	5	TIMER - set RTC date
	6	TIMER - set RTC alarm
	7	TIMER - delete RTC alarm
	8	TIMER - RTC activated power on
	9	TIMER - read RTC alarm and status
	0A	TIMER - get system counter (days)
	0B	TIMER - set system counter (days)
	0C-7F	RESERVED
	80	SOUND - multiplexer
	81-FF	RESERVED
$21	0	terminate program
	1	read and output characters from keyboard
	2	display characters
	3	receive character from serial port
	4	send characters to serial port
	5	send characters to parallel port
	6	direct character I/O
	7	direct character input without echo
	8	read characters from keyboard
	9	print characters
	A	buffered string input
	B	get keyboard status
	C	delete input buffer and invoke input
	D	drive reset
	E	select drive
	F	open file
	10	close file
	11	search for first FCB entry
	12	search for next FCB entry
	13	delete file
	14	sequential read
	15	sequential write

DOS Interrupts

Interrupt	Function	Description
$21	16	create file
	17	rename file
	18	RESERVED
	19	return current drive
	1A	set DTA (Disk Transfer Address)
	1B	return allocation for current drive
	1C	return allocation for specific drive
	1D	RESERVED
	1E	RESERVED
	1F	RESERVED
	20	RESERVED
	21	random read
	22	random write
	23	get file size
	24	set record number
	25	set interrupt vector
	26	create new program segment
	27	random block read
	28	random block write
	29	parse filename
	2A	get date
	2B	set date
	2C	get time
	2D	set time
	2E	set verify flag
	2F	return DTA
	30	get DOS version number
	31	terminate program and stay resident
	32	RESERVED
	33	get/set Control-C status
	34	RESERVED
	35	get interrupt vector

DOS Interrupts

Interrupt	Function	Description
$21	36	get disk free space
	37	RESERVED
	38,0	get country dependent information
	38,X	set country dependent information (X=country code)
	39	create subdirectory
	3A	remove subdirectory
	3B	change directory
	3C	create file (handle)
	3D	open file (handle)
	3E	close file (handle)
	3F	read from file or device (handle)
	40	write to file or device (handle)
	41	delete file (handle)
	42	move file pointer
	43	get/set file attribute
	44,0	IOCTL - get device attribute
	44,1	IOCTL - set device attribute
	44,2	IOCTL - receive characters from driver
	44,3	IOCTL - send characters to driver
	44,4	IOCTL - receive character from block driver
	44,5	IOCTL - send characters to block driver
	44,6	IOCTL - return input status
	44,7	IOCTL - return output status
	44,8	IOCTL - is medium changeable?
	44,9	IOCTL - remote device test
	44,A	IOCTL - remote handle test
	44,B	IOCTL - change retry count
	44,C	IOCTL - handles
	44,D	IOCTL - devices
	44,E	get drive
	44,F	set drive
	45	duplicate a file handle

DOS Interrupts

Interrupt	Function	Description
$21	46	force same file handle
	47	get current directory
	48	allocate memory
	49	free allocated memory
	4A	set allocate memory block
	4B,0	program load and execute
	4B,3	load overlay
	4C	terminate process
	4D	return end code from a process
	4E	find first file entry
	4F	find next file entry
	50	RESERVED
	51	RESERVED
	52	RESERVED
	53	RESERVED
	54	return verify status
	55	RESERVED
	56	rename file
	57	get/set file time/date stamp
	58	get/set file recording mode
	59	get extended error
	5A	create temporary file
	5B	create new file
	5C,0	lock access
	5C,1	unlock access
	5D	RESERVED
	5E,0	return machine name
	5E,2	printer setting
	5F,2	read entry from allocation table
	5F,3	insert entry into allocation table
	5F,4	delete entry from allocation table
	60	RESERVED

DOS Interrupts

Interrupt	Function	Description
$21	61	RESERVED
	62	return PSP address
	63	get byte table
	65	read extended country dependent information
	66	get/set code page
	67	set handle count
	68	deliver file
$24	-	critical error handler vector
$25	-	read data from disk (absolute)
$26	-	write data to disk (absolute)
$33	-	mouse interrupt
$67	-	EMS interrupt

Reference: Programmer's Guide to the IMB PC, (Microsoft Press), Peter Norton, chapter 8 to 13
IBM PC/XT Technical Reference BIOS Listings
IBM PC/AT Technical Reference BIOS Listings
IBM PS/2 and PC BIOS Interface Technical Reference, pages 2-10 to 2-122
IBM DOS 3.3 Technical Reference, pages 6-1 to 6-33 and 6-6 to 6-7

Index

+ .... 2 - 28
< .... 2 - 28
< = ... 2 - 28
< > .... 2 - 28
= .... 2 - 28
= > .... 2 - 28
> .... 2 - 28

_AH .... 4 - 5, 4 - 8
_AX .... 4 - 14
_BX .... 4 - 14
_CX .... 4 - 14
_DX .... 4 - 14
_X .... 2 - 115, 2 - 119
_Y .... 2 - 115, 2 - 119

3D effects .... 3 - 3

ALERT .... 1 - 26, 3 - 16
Alert boxes .... 3 - 16
Area Cosinus hyperbolicus .... 1 - 33
Area Cotangens hyperbolicus .... 1 - 34
Area Sinus hyperbolicus .... 1 - 33
Area Tangens hyperbolicus .... 1 - 33
Arithmetic mean .... 1 - 29
ARRAYFILL .... 2 - 40

BIN$() .... 2 - 27
BIOS .... 4 - 1
BMOVE .... 1 - 28
Boolean .... 2 - 3
BOX .... 2 - 115, 2 - 118, 2 - 121, 2 - 127
Bytes .... 2 - 2

Call by reference' variables.......... 1 - 22
Call by value' variables.......... 1 - 22, 1 - 26
CASE .... 1 - 15
CGA resolution.......... 2 - 107
CGA resolution.......... 3 - 4
Character graphics.......... 2 - 106
Character output.......... 4 - 8
CHR$() .... 2 - 27
CIRCLE .... 2 - 115, 2 - 121, 2 - 129
CLEARW.......... 3 - 34
CLIP .... 2 - 116, 2 - 136, 2 - 137
Closer .... 3 - 29
CLOSEW.......... 3 - 32, 3 - 33, 3 - 56
COLOR .... 2 - 105, 2 - 115, 2 - 127, 2 - 128, 2 - 130, 3 - 27
Colour graphic card (CGA).......... 2 - 104
Column matrices.......... 2 - 41
Combining strings.......... 2 - 14
Conditional statements.......... 1 - 1, 1 - 10
Configuration parameters.......... 3 - 52
CONT .... 1 - 15
Copy and exchange commands.......... 2 - 35, 2 - 48
Cosinus hyperbolicus.......... 1 - 33
Cotangens hyperbolicus.......... 1 - 33
CURVE .... 2 - 116, 2 - 131

Debug variables.......... 1 - 25
DEC$() .... 2 - 27
DEFAULT.......... 1 - 15
DEFFILL.......... 2 - 115, 2 - 119, 2 - 130
DEFLINE.......... 2 - 115, 2 - 121, 2 - 122
DEFSTR.......... 2 - 26
DEFTEXT.......... 2 - 116
Desktop .... 3 - 27
Diagonal matrix.......... 2 - 42, 2 - 43
DO...LOOP loop.......... 1 - 6, 1 - 8, 3 - 20

DOS .... 4 - 1
Double .... 2 - 1
DRAW .... 2 - 115, 2 - 123

EAVAIL .... 2 - 140
EDIR .... 2 - 140
EGA card .... 2 - 114, 3 - 4
ELLIPSE .... 2 - 115, 2 - 121, 2 - 130
ELSE .... 1 - 11
Empirical median .... 1 - 30
Empirical median deviation .... 1 - 32
Empirical standard deviation .... 1 - 31
Empirical standard error .... 1 - 32
Empirical variation .... 1 - 31
EMSGET .... 2 - 116, 2 - 140
EMSPUT .... 2 - 116, 2 - 140
ENDIF .... 1 - 11
ENDSELECT .... 1 - 15
Evaluation functions .... 2 - 33
EXIT IF .... 1 - 4, 1 - 8
Expansions .... 4 - 1

FILL .... 2 - 116, 2 - 118, 2 - 133
Floating point operators .... 2 - 5
FOR...NEXT loop .... 1 - 2
FULLW .... 3 - 32, 3 - 36, 3 - 56
FUNCTION .... 1 - 20

Generating commands .... 2 - 35, 2 - 39
Geometric mean .... 1 - 29
GET .... 2 - 116, 2 - 138, 2 - 139
GETEVENT .... 3 - 5, 3 - 33
GETFIRST .... 3 - 56, 3 - 57
GETNEXT .... 3 - 56, 3 - 58
Global variables .... 1 - 25
Graphic interface .... 3 - 1
Graphic mode .... 3 - 3

Graphics .... 2 - 103
GRAPHMODE .... 2 - 115, 2 - 117

Harmonic mean .... 1 - 30
HEX$() .... 2 - 27
Hyperbolean functions .... 1 - 33

IF...ENDIF conditional statement .... 1 - 10, 1 - 11, 3 - 20
INFOW .... 3 - 32, 3 - 41
Initialisation .... 4 - 5
INPUT .... 1 - 27
Inquire printer status .... 4 - 11
Inquire status .... 4 - 6
INSTR() .... 2 - 28, 2 - 33, 4 - 1

Joystick .... 4 - 13
Joysticks position .... 4 - 14

LCASE$() .... 2 - 27, 2 - 29
LEFT$() .... 2 - 27, 2 - 29, 2 - 30
LINE .... 2 - 115, 2 - 121, 2 - 122
LOCAL .... 1 - 24
Local variables .... 1 - 25
Logical operators .... 2 - 8
LOGO .... 2 - 124
Long .... 2 - 2
Loops .... 1 - 1, 1 - 2
Lower triangular matrix .... 2 - 42
LSET .... 2 - 28

Manipulation functions .... 2 - 29
MAT ADD .... 2 - 39, 2 - 57
MAT BASE .... 2 - 35
MAT CLR .... 2 - 39, 2 - 40
MAT commands .... 2 - 35
MAT CPY .... 2 - 39, 2 - 48
MAT DET .... 2 - 57, 2 - 78

MAT DIAG .... 2 - 39, 2 - 40, 2 - 44
MAT INPUT .... 2 - 45, 2 - 47
MAT INV .... 2 - 57, 2 - 85
MAT MUL .... 2 - 39, 2 - 57, 2 - 63
MAT NEG .... 2 - 39, 2 - 40
MAT NORM .... 2 - 57, 2 - 72
MAT ONE .... 2 - 39, 2 - 40, 2 - 43
MAT PRINT .... 2 - 39, 2 - 45, 2 - 46
MAT QDET .... 2 - 57, 2 - 80
MAT RANK .... 2 - 57, 2 - 81
MAT READ .... 2 - 39, 2 - 45
MAT SET .... 2 - 39, 2 - 40
MAT SUB .... 2 - 39, 2 - 57, 2 - 60
MAT TRANS .... 2 - 48, 2 - 55
MAT TRI .... 2 - 39, 2 - 40, 2 - 44
MAT XCPY .... 2 - 39, 2 - 48, 2 - 53
Matrices .... 2 - 41
Maximizer .... 3 - 29
MDA resolution .... 2 - 106
MENU .... 3 - 1
MENU() .... 3 - 1, 3 - 6, 3 - 44
MID$() .... 2 - 27, 2 - 29, 2 - 30
Minimizer .... 3 - 29
MIRROR$() .... 2 - 27, 2 - 29, 2 - 32
MKD$() .... 2 - 27
MKF$() .... 2 - 27
MKI$() .... 2 - 27
MKL$() .... 2 - 27
MKS$() .... 2 - 27
Monochrome (Hercules) graphic card (HGA) .... 2 - 104
Monochrome (IBM) video card (MDA) .... 2 - 104
MOUSEX .... 3 - 40
MOUSEY .... 3 - 40
Mover .... 3 - 29
MOVEW .... 3 - 32, 3 - 34, 3 - 56

Numeric arrays .... 2 - 41
Numerical expressions .... 2 - 23

OCT$() .... 2 - 27
OFFSET .... 2 - 133
ON MENU .... 3 - 5, 3 - 6
OPENW # .... 3 - 3, 3 - 27, 3 - 41, 3 - 52, 3 - 56
Operating commands .... 2 - 35, 2 - 57
Operator hierarchy .... 2 - 17
Operators .... 2 - 5
OPTION BASE .... 2 - 35, 3 - 3
Output characters .... 4 - 12

PBOX .... 2 - 115, 2 - 129, 3 - 39
PCIRCLE .... 2 - 115, 2 - 118, 2 - 130
PEEKEVENT .... 3 - 5
PELLIPSE .... 2 - 116, 2 - 118, 2 - 131
Pixel graphic card .... 2 - 113
PLOT .... 2 - 115, 2 - 119, 2 - 123
POLYFILL .... 2 - 116, 2 - 118, 2 - 132
POLYLINE .... 2 - 116, 2 - 121, 2 - 132
Pop-up menus .... 3 - 3, 3 - 16
POPUP .... 3 - 21
PRBOX .... 2 - 115
PRED() .... 2 - 28, 2 - 33
PRINT .... 1 - 27
Printer (parallel) port .... 4 - 10
Printer initialisation .... 4 - 10
Printer (serial) port .... 4 - 5
PROCEDURE .... 1 - 20
PSET .... 2 - 115, 2 - 120
Pull-down menus .... 3 - 3
PUT .... 2 - 116, 2 - 139

Quadratic (square) matrix .... 2 - 41

RBOX .... 2 - 115, 2 - 121, 2 - 128
RC_INTERSECT .... 3 - 56, 3 - 57
Read and write commands .... 2 - 35, 2 - 45
Read character .... 4 - 2, 4 - 7
Read disk sectors .... 4 - 15
Redraw events .... 3 - 56
Relational operators .... 2 - 14
REPEAT...UNTIL .... 1 - 6, 1 - 7
Report shift status .... 4 - 3
RETURN .... 1 - 21
Reverse area functions .... 1 - 33
RIGHT$() .... 2 - 27, 2 - 29, 2 - 30
RINSTR() .... 2 - 28, 2 - 33, 2 - 34
Row matrices .... 2 - 41
RSET .... 2 - 28
Rules of matrix addition .... 2 - 58
Rules of matrix multiplication .... 2 - 63
Rules of matrix subtraction .... 2 - 60

SCREEN .... 2 - 104, 2 - 105, 2 - 114, 2 - 115
SCROLL ON/OFF .... 2 - 106, 2 - 112
SELECT...CASE .... 1 - 10, 1 - 14, 1 - 15
SELECT...CASE conditional statement .... 1 - 16
Set the colours .... 3 - 3
SETDRAW .... 2 - 115, 2 - 126
Single line functions .... 1 - 33
Sinus hyperbolicus .... 1 - 33
SIZEW .... 3 - 32, 3 - 56
Sliders .... 3 - 53
SPACE$() .... 2 - 27, 2 - 29, 2 - 31
Square matrix .... 2 - 42, 2 - 43
Statistical values .... 1 - 28
Status of special keys .... 4 - 3
Status of the fire button .... 4 - 13
STEP .... 1 - 3
STR$() .... 2 - 27
String .... 2 - 3

String operators 2 - 14, 2 - 18, 2 - 28
Structured programming 1 - 1
Subroutines 1 - 1
SUCC() 2 - 28, 2 - 33
Symmetrical matrix 2 - 43
SYSCOL 3 - 1, 3 - 27
System command 2 - 35
System routine 4 - 1

Tangens hyperbolicus 1 - 33
TBOX 2 - 106, 2 - 109, 2 - 110
TCLIP 2 - 106, 2 - 110
TCOLOR 2 - 104, 2 - 106, 2 - 110
TEXT 1 - 27, 2 - 116, 2 - 134
Text graphic cards 2 - 103
TGET 2 - 106, 2 - 112
TITLEW 3 - 32, 3 - 41
TOPW 3 - 56
TPBOX 2 - 106, 2 - 110
TPUT 2 - 106, 2 - 112
TRIM$() 2 - 27, 2 - 29, 2 - 31
TYPE 1 - 25
Type conversion 2 - 18

UCASE$() 2 - 27, 2 - 29, 2 - 31
Uniform matrix 2 - 43
Upper triangular matrix 2 - 42
UPPER$() 2 - 27, 2 - 29, 2 - 31
Using strings 2 - 26

VAR 1 - 24
Vectors 2 - 41
VGA card 2 - 114
VGA resolution 3 - 4

Waiting loop........ 1 - 19
WHILE...WEND loop........ 1 - 6
WIN .... 3 - 57
WINDFIND........ 3 - 55
WINDGET........ 3 - 54, 3 - 57
Windows........ 3 - 3, 3 - 27
WINDSET........ 3 - 53
Word .... 2 - 2
Word lists........ 1 - 26
WRAP ON/OFF........ 2 - 106, 2 - 111
Writing subroutines........ 1 - 19

XLATE$()........ 2 - 27, 2 - 29, 2 - 32

GFA Systemtechnik GmbH
Heerdter Sandberg 30
D-4000 Düsseldorf 11
Telefon 0211/5504-0

GFA
SYSTEMTECHNIK